O'REILLY®

C# 10 in a Nutshell

The Definitive Reference



Joseph Albahari

Prefacio

C# 10 representa la novena actualización importante del lenguaje de programación estrella de Microsoft, posicionando a C# como un lenguaje con una flexibilidad y amplitud inusual. Por un lado, ofrece abstracciones de alto nivel como expresiones de consulta y continuaciones asíncronas, mientras que por otro lado, permite una eficiencia de bajo nivel mediante construcciones como tipos de valor personalizados y punteros opcionales.

El precio de este crecimiento es que hay más que nunca por aprender. Aunque herramientas como IntelliSense de Microsoft y referencias en línea son excelentes para ayudarte en el trabajo, presuponen un mapa existente de conocimientos conceptuales. Este libro proporciona precisamente ese mapa de conocimiento de manera concisa y unificada, sin desorden y sin introducciones largas.

Al igual que las siete ediciones anteriores, C# 10 en pocas palabras está organizado en torno a conceptos y casos de uso, lo que lo hace amigable tanto para la lectura secuencial como para la exploración aleatoria. También profundiza en temas significativos mientras asume solo conocimientos básicos, lo que lo hace accesible tanto para lectores intermedios como avanzados.

Este libro abarca C#, el Common Language Runtime (CLR) y la Base Class Library (BCL) de .NET 6. Hemos elegido este enfoque para permitir espacio para temas difíciles y avanzados sin comprometer la profundidad o la legibilidad. Las características agregadas recientemente a C# se indican para que también puedas usar este libro como referencia para C# 9, C# 8 y C# 7.

Público objetivo

Este libro está dirigido a audiencias intermedias y avanzadas. No se requiere conocimiento previo de C#, pero se necesita cierta experiencia en programación general. Para principiantes, este libro complementa, en lugar de reemplazar, una introducción al estilo de tutorial a la programación.

Este libro es un compañero ideal para cualquiera de la amplia gama de libros que se centran en una tecnología aplicada, como ASP.NET Core o Windows Presentation Foundation (WPF). C# 10 en pocas palabras cubre las áreas del lenguaje y .NET que estos libros omiten, y viceversa.

Si estás buscando un libro que cubra todas las tecnologías de .NET superficialmente, esto no es para ti. Este libro también es inadecuado si deseas aprender sobre APIs específicas para el desarrollo de dispositivos móviles.

Organización del libro

Los capítulos 2 al 4 se centran exclusivamente en C#, comenzando con los conceptos básicos de sintaxis, tipos y variables, y terminando con temas avanzados como código inseguro y directivas del preprocesador. Si eres nuevo en el lenguaje, debes leer estos capítulos de manera secuencial.

Los capítulos restantes se centran en las bibliotecas de clases base (BCLs) de .NET 6, cubriendo temas como Language-Integrated Query (LINQ), XML, colecciones, concurrencia, E/S y redes, gestión de memoria, reflexión, programación dinámica, atributos, criptografía e interoperabilidad nativa. Puedes leer la mayoría de estos capítulos al azar, excepto los Capítulos 5 y 6, que sientan las bases para los temas posteriores. También es recomendable leer los tres

capítulos sobre LINQ de manera secuencial, y algunos capítulos asumen cierto conocimiento de concurrencia, que se cubre en el Capítulo 14.

Lo que necesitas para utilizar este libro

Los ejemplos de este libro requieren .NET 6. También encontrarás útil la documentación de .NET de Microsoft para consultar tipos y miembros individuales (disponible en línea).

Aunque es posible escribir código fuente en un editor de texto simple y compilar el programa desde la línea de comandos, serás mucho más productivo con un bloc de notas de código para probar instantáneamente fragmentos de código, además de un entorno de desarrollo integrado (IDE) para producir ejecutables y bibliotecas.

Para un bloc de notas de código en Windows, descarga LINQPad 7 de www.linqpad.net (gratuito). LINQPad es compatible completamente con C# 10 y está mantenido por el autor.

Para un IDE en Windows, descarga Visual Studio 2022: cualquier edición es adecuada para lo que se enseña en este libro. Para un IDE multiplataforma, descarga Visual Studio Code.

Convenciones utilizadas en este libro

El libro utiliza la notación básica de UML para ilustrar las relaciones entre tipos, como se muestra en la Figura P-1. Un rectángulo inclinado representa una clase abstracta; un círculo representa una interfaz. Una línea con un triángulo hueco denota herencia, con el triángulo apuntando al tipo base. Una línea con una flecha denota una asociación unidireccional; una línea sin flecha denota una asociación bidireccional.

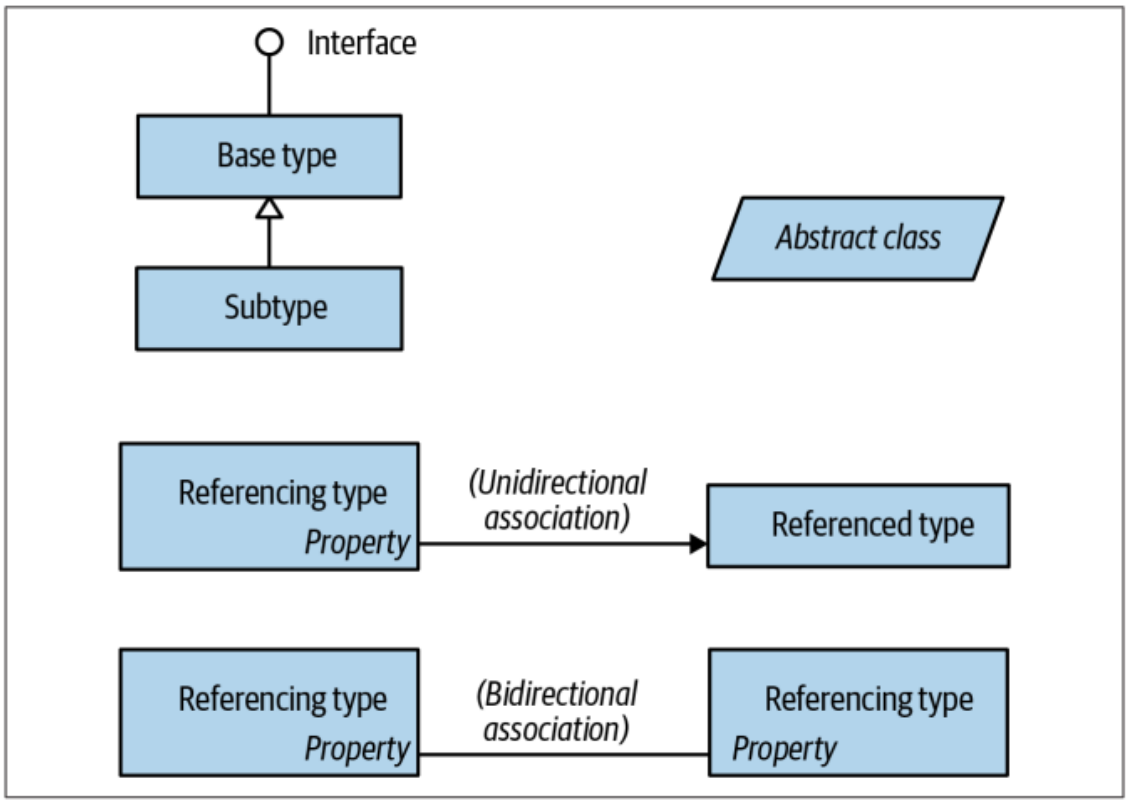


Figure P-1. Sample diagram

1 Introducción a C# y .NET

C# es un lenguaje de programación orientado a objetos, general y seguro en cuanto a tipos. El objetivo del lenguaje es la productividad del programador. Para lograr esto, C# equilibra la simplicidad, la expresividad y el rendimiento. El arquitecto principal del lenguaje desde su primera versión es Anders Hejlsberg (creador de Turbo Pascal y arquitecto de Delphi).

El lenguaje C# es neutral en cuanto a la plataforma y funciona con una variedad de entornos de ejecución específicos de la plataforma.

Orientación a objetos

C# es una implementación completa del paradigma de la orientación a objetos, que incluye encapsulación, herencia y polimorfismo. La encapsulación significa crear un límite alrededor de un objeto para separar su comportamiento externo (público) de los detalles de implementación internos (privados). A continuación, se presentan las características distintivas de C# desde una perspectiva orientada a objetos:

Sistema de tipos unificado

El bloque básico fundamental en C# es una unidad encapsulada de datos y funciones llamada tipo. C# tiene un sistema de tipos unificado en el que todos los tipos comparten en última instancia un tipo base común. Esto significa que todos los tipos, ya sea que representen objetos de negocio o sean tipos primitivos como números, comparten la misma funcionalidad básica. Por ejemplo, una instancia de cualquier tipo se puede convertir a una cadena llamando a su método ToString.

Clases e interfaces

En un paradigma orientado a objetos tradicional, el único tipo existente es una clase. En C#, hay varios otros tipos, uno de los cuales es una interfaz. Una interfaz es similar a una clase, pero no puede contener datos. Esto significa que solo puede definir comportamiento (y no estado), lo que permite la herencia múltiple y la separación entre especificación e implementación.

Propiedades, métodos y eventos

En el paradigma puramente orientado a objetos, todas las funciones son métodos. En C#, los métodos son solo una forma de miembro de función, que también incluye propiedades y eventos (también hay otros). Las propiedades son miembros de función que encapsulan una parte del estado de un objeto, como el color de un botón o el texto de una etiqueta. Los eventos son miembros de función que simplifican la acción de los cambios de estado del objeto.

Aunque C# es principalmente un lenguaje orientado a objetos, también toma prestado del paradigma de programación funcional, específicamente:

Las funciones se pueden tratar como valores

Mediante el uso de delegados, C# permite que las funciones se pasen como valores hacia y desde otras funciones.

C# admite patrones de pureza

En el núcleo de la programación funcional está evitar el uso de variables cuyos valores cambian, a favor de patrones declarativos. C# tiene características clave para ayudar con esos patrones, incluida la capacidad de escribir funciones sin nombre sobre la marcha que "capturan" variables (expresiones lambda) y la capacidad de realizar programación de listas o reactiva a través de expresiones de consulta. C# también proporciona registros, que facilitan la escritura de tipos inmutables (de solo lectura).

Seguridad de tipos

C# es principalmente un lenguaje seguro en cuanto a tipos, lo que significa que las instancias de los tipos solo pueden interactuar a través de los protocolos que definen, asegurando así la coherencia interna de cada tipo. Por ejemplo, C# te impide interactuar con un tipo de cadena como si fuera un tipo entero.

Más específicamente, C# admite la tipificación estática, lo que significa que el lenguaje impone la seguridad de tipos en tiempo de compilación. Esto se suma a la seguridad de tipos que se aplica en tiempo de ejecución. La tipificación estática elimina una gran cantidad de errores antes de que se ejecute un programa. Desplaza la carga de las pruebas unitarias en tiempo de ejecución al compilador para verificar que todos los tipos en un programa encajen correctamente. Esto hace que los programas grandes sean mucho más fáciles de gestionar, más predecibles y más robustos. Además, la tipificación estática permite que herramientas como IntelliSense en Visual Studio te ayuden a escribir un programa porque saben qué tipo tiene una variable y, por lo tanto, qué métodos se pueden llamar en esa variable. Estas herramientas también pueden identificar en todo tu programa dónde se utiliza una variable, tipo o método, lo que permite refactorización confiable.



C# también permite que partes de tu código tengan un tipo dinámico mediante el uso de la palabra clave "dynamic". Sin embargo, C# sigue siendo predominantemente un lenguaje de tipado estático.

C# también se considera un lenguaje de tipado fuerte porque sus reglas de tipos se aplican estrictamente, ya sea en tiempo de compilación o en tiempo de ejecución. Por ejemplo, no puedes llamar a una función que está diseñada para aceptar un número entero con un número de punto flotante, a menos que conviertas explícitamente el número de punto flotante a un número entero. Esto ayuda a prevenir errores.

Gestión de memoria

C# se basa en el tiempo de ejecución para realizar la gestión automática de memoria. El Common Language Runtime cuenta con un recolector de basura que se ejecuta como parte de tu programa, recuperando la memoria de los objetos que ya no están referenciados. Esto libera a

los programadores de tener que liberar explícitamente la memoria de un objeto, eliminando el problema de los punteros incorrectos que se encuentran en lenguajes como C++.

C# no elimina los punteros, simplemente los hace innecesarios para la mayoría de las tareas de programación. Para áreas críticas de rendimiento e interoperabilidad, se permiten punteros y asignación explícita de memoria en bloques que se marcan como inseguros.

Soporte de plataformas

C# cuenta con tiempos de ejecución que admiten las siguientes plataformas:

• Windows Desktop 7-11 (para aplicaciones de cliente rico, web, servidor y de línea de comandos)

• macOS (para aplicaciones de cliente rico, web y de línea de comandos)

• Linux y macOS (para aplicaciones web y de línea de comandos)

• Android e iOS (para aplicaciones móviles)

• Dispositivos con Windows 10 (Xbox, Surface Hub y HoloLens)

También existe una tecnología llamada Blazor que puede compilar C# a ensamblado web que se ejecuta en un navegador.

CLRs, BCLs y tiempos de ejecución

El soporte en tiempo de ejecución para programas de C# consta de un Common Language Runtime y una Base Class Library. Un tiempo de ejecución también puede incluir una capa de aplicación de nivel superior que contiene bibliotecas para desarrollar aplicaciones de cliente rico, móviles o web (ver Figura 1-1). Existen diferentes tiempos de ejecución para permitir diferentes tipos de aplicaciones, así como diferentes plataformas.

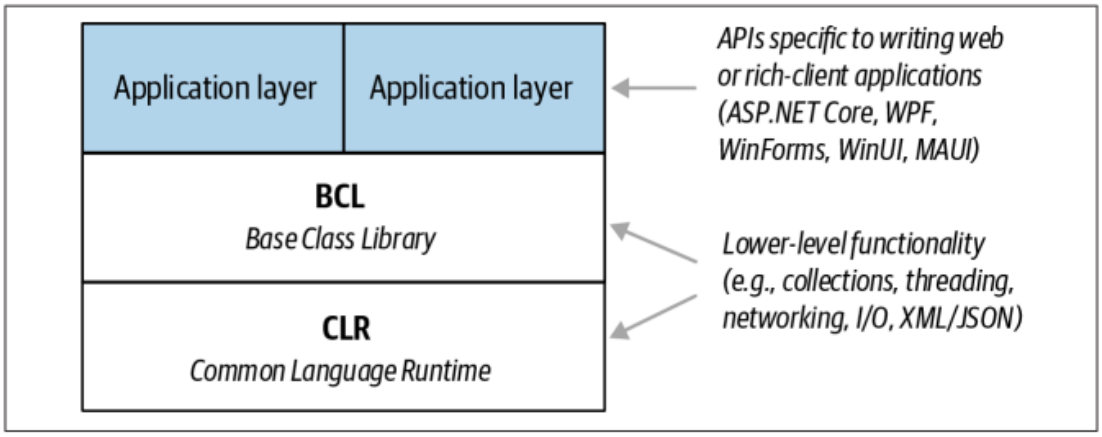


Figure 1-1. Runtime architecture

Tiempo de ejecución de lenguaje común (Common Language Runtime)

Un tiempo de ejecución de lenguaje común (Common Language Runtime o CLR) proporciona servicios fundamentales en tiempo de ejecución, como la gestión automática de memoria y el manejo de excepciones. (El término "común" se refiere al hecho de que el mismo tiempo de ejecución puede ser compartido por otros lenguajes de programación gestionados, como F#, Visual Basic y Managed C++).

C# se llama un lenguaje gestionado porque compila el código fuente en código gestionado, que se representa en Lenguaje Intermedio (Intermediate Language o IL). El CLR convierte el IL en el código nativo de la máquina, como X64 o X86, generalmente justo antes de la ejecución. Esto se conoce como compilación Just-in-Time (JIT). También está disponible la compilación previa al tiempo de ejecución para mejorar el tiempo de inicio con ensamblados grandes o dispositivos con limitaciones de recursos (y para cumplir con las reglas de la tienda de aplicaciones de iOS al desarrollar aplicaciones móviles).

El contenedor para el código gestionado se llama ensamblado. Un ensamblado contiene no solo IL, sino también información de tipos (metadatos). La presencia de metadatos permite que los ensamblados hagan referencia a tipos en otros ensamblados sin necesidad de archivos adicionales.



Puedes examinar y desensamblar el contenido de un ensamblado con la herramienta "ildasm" de Microsoft. Y con herramientas como ILSpy o dotPeek de JetBrains, puedes ir más allá y descompilar el IL a C#. Debido a que el IL es de un nivel más alto que el código de máquina nativo, el descompilador puede hacer un buen trabajo al reconstruir el código C# original.

Un programa puede consultar su propia metadata (reflexión) e incluso generar nuevo IL en tiempo de ejecución (reflection.emit).

Biblioteca Base de Clases (BCL)

Un CLR siempre se envía con un conjunto de ensamblados llamado Biblioteca Base de Clases (BCL). La BCL proporciona funcionalidad básica a los programadores, como colecciones, entrada/salida, procesamiento de texto, manejo de XML/JSON, redes, encriptación, interoperabilidad, concurrencia y programación paralela.

La BCL también implementa tipos que el propio lenguaje C# requiere (para características como enumeración, consultas y asincronía) y te permite acceder explícitamente a características del CLR, como reflexión y gestión de memoria.

Runtimes

Un tiempo de ejecución (también llamado framework) es una unidad desplegable que se descarga e instala. Un tiempo de ejecución consta de un CLR (con su BCL), además de una capa de aplicación opcional específica para el tipo de aplicación que estás escribiendo, ya sea web, móvil, cliente rico, etc. (Si estás escribiendo una aplicación de consola de línea de comandos o una biblioteca sin interfaz de usuario, no necesitas una capa de aplicación).

Cuando escribes una aplicación, apuntas a un tiempo de ejecución específico, lo que significa que tu aplicación utiliza y depende de la funcionalidad que proporciona el tiempo de ejecución.

La elección de tu entorno de ejecución también determina qué plataformas admitirá tu aplicación.

La siguiente tabla enumera las principales opciones de entorno de ejecución:

Application layer	CLR/BCL	Program type	Runs on...
ASP.NET	.NET 6	Web	Windows, Linux, macOS
Windows Desktop	.NET 6	Windows	Windows 7–10+
MAUI (early 2022)	.NET 6	Mobile, desktop	iOS, Android, macOS, Windows 10+
WinUI 3 (early 2022)	.NET 6	Win10	Windows 10+ desktop
UWP	.NET Core 2.2	Win10 + Win10 devices	Windows 10+ desktop & devices
(Legacy) .NET Framework	.NET Framework	Web, Windows	Windows 7–10+

La Figura 1-2 muestra esta información de forma gráfica y también sirve como guía de lo que se cubre en el libro.

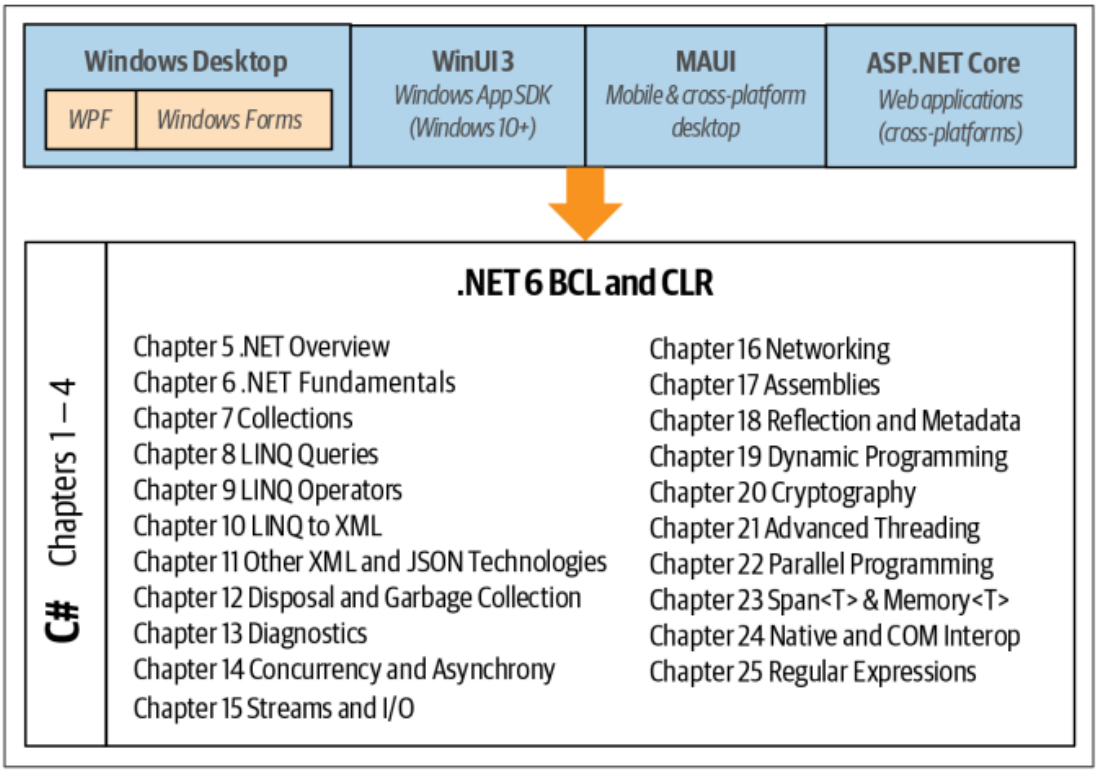


Figure 1-2. Runtimes for C#

.NET 6

.NET 6 es el tiempo de ejecución de código abierto principal de Microsoft. Puedes escribir aplicaciones web y de consola que se ejecutan en Windows, Linux y macOS; aplicaciones de cliente con interfaz gráfica que se ejecutan en Windows 7 a 11 y macOS; y aplicaciones móviles que se ejecutan en iOS y Android. Este libro se centra en el tiempo de ejecución .NET 6 y la Biblioteca de Clases Base (BCL).

A diferencia de .NET Framework, .NET 6 no viene preinstalado en las máquinas con Windows. Si intentas ejecutar una aplicación de .NET 6 sin tener el tiempo de ejecución correcto, aparecerá un mensaje que te dirigirá a una página web donde podrás descargar el tiempo de ejecución. Puedes evitar esto creando una implementación autocontenida, que incluye las partes del tiempo de ejecución requeridas por la aplicación.



El predecesor de .NET 6 fue .NET 5, cuyo predecesor fue .NET Core 3 (Microsoft eliminó "Core" del nombre y omitió la versión 4). La razón para omitir una versión fue evitar confusiones con .NET Framework 4.x.

Esto significa que los ensamblados compilados bajo las versiones 1, 2 y 3 de .NET Core (y .NET 5), en la mayoría de los casos, se ejecutarán sin modificaciones en .NET 6. En contraste, los ensamblados compilados bajo (cualquier versión de) .NET Framework generalmente son incompatibles con .NET 6.

El BCL y el CLR de .NET 6 son muy similares a .NET 5 (y .NET Core 3), con diferencias principalmente en rendimiento y despliegue.

MAUI

MAUI (Multi-platform App UI, principios de 2022) está diseñado para crear aplicaciones móviles para iOS y Android, así como aplicaciones de escritorio multiplataforma para macOS y Windows. MAUI es una evolución de Xamarin y permite que un único proyecto se oriente a múltiples plataformas.

UWP and WinUI 3

UWP (Universal Windows Platform) está diseñado para escribir aplicaciones táctiles inmersivas que se ejecutan en Windows 10+ en escritorios y dispositivos (Xbox, Surface Hub y HoloLens). Las aplicaciones UWP están aisladas y se distribuyen a través de la Tienda de Windows. UWP se encuentra preinstalado en Windows 10 y utiliza una versión de .NET Core 2.2 CLR/BCL, y es poco probable que esta dependencia se actualice. En su lugar, Microsoft ha lanzado un sucesor llamado WinUI 3 como parte del Windows App SDK.

El Windows App SDK funciona con la última versión de .NET, se integra mejor con las API de escritorio de .NET y puede ejecutarse fuera de un entorno aislado. Sin embargo, aún no es compatible con dispositivos como Xbox o HoloLens.

.NET Framework

.NET Framework es el tiempo de ejecución original de Microsoft exclusivo de Windows para escribir aplicaciones web y de cliente enriquecidas que se ejecutan solo en equipos de escritorio/servidores Windows. No se planean importantes lanzamientos nuevos, aunque Microsoft seguirá brindando soporte y mantenimiento para la versión actual 4.8 debido a la gran cantidad de aplicaciones existentes.

Con .NET Framework, el CLR/BCL está integrado con la capa de aplicación. Las aplicaciones escritas en .NET Framework pueden ser recompiladas en .NET 6, aunque generalmente requieren algunas modificaciones. Algunas características de .NET Framework no están presentes en .NET 6 (y viceversa).

.NET Framework se encuentra preinstalado en Windows y se actualiza automáticamente a través de Windows Update. Cuando te orientas a .NET Framework 4.8, puedes utilizar las características de C# 7.3 y versiones anteriores.



La palabra ".NET" ha sido utilizada durante mucho tiempo como un término general para cualquier tecnología que incluya la palabra ".NET" (como .NET Framework, .NET Core, .NET Standard, entre otros).

Esto significa que el cambio de nombre de .NET Core a .NET ha creado una ambigüedad desafortunada. En este libro, nos referiremos al nuevo .NET como ".NET 5+". Y para referirnos a .NET Core y sus sucesores, utilizaremos la frase ".NET Core y .NET 5+".

Para añadir a la confusión, .NET (5+) es un marco de trabajo, pero es muy diferente del .NET Framework. Por lo tanto, utilizaremos el término "tiempo de ejecución" en lugar de "marco de trabajo" siempre que sea posible.

Tiempo de ejecución de nicho

También existen los siguientes tiempos de ejecución de nicho:

• El .NET Micro Framework está diseñado para ejecutar código .NET en dispositivos integrados con recursos muy limitados (menos de un megabyte).

• Unity es una plataforma de desarrollo de juegos que permite el uso de lógica de juegos escrita en C#.

También es posible ejecutar código administrado dentro de SQL Server. Con la integración de CLR de SQL Server, puedes escribir funciones personalizadas, procedimientos almacenados y agregaciones en C# y luego llamarlas desde SQL. Esto funciona en conjunto con .NET Framework y un CLR "alojado" especial que aplica un entorno seguro para proteger la integridad del proceso de SQL Server.

Breve historia de C#

A continuación, se muestra una cronología inversa de las nuevas características de cada versión de C#, para beneficio de los lectores que ya están familiarizados con una versión anterior del lenguaje.

Novedades en C# 10

C# 10 se incluye en Visual Studio 2022 y se utiliza al dirigirse a .NET 6.

Espacios de nombres con ámbito de archivo

En el caso común de que todos los tipos en un archivo estén definidos en un solo espacio de nombres, una declaración de espacio de nombres con ámbito de archivo en C# 10 reduce el desorden y elimina un nivel innecesario de sangría:

```
namespace MyNamespace;  // Applies to everything that follows in the file.

class Class1 {}         // inside MyNamespace
class Class2 {}         // inside MyNamespace
```

La directiva global using

Cuando se utiliza la palabra clave global antes de una directiva using, esta se aplica a todos los archivos del proyecto:

```
global using System;
global using System.Collection.Generic;
```

Esto te permite evitar repetir las mismas directivas en cada archivo. Las directivas global using también funcionan con using static.

Además, en los proyectos de .NET 6 ahora se admiten directivas globales using implícitas: si el elemento ImplicitUsings está establecido en true en el archivo del proyecto, se importan automáticamente los espacios de nombres más utilizados (según el tipo de proyecto SDK). Consulta "La directiva global using (C# 10)" en la página 88 para obtener más detalles.

Mutación no destructiva para tipos anónimos

C# 9 introdujo la palabra clave with para realizar mutaciones no destructivas en registros. En C# 10, la palabra clave with también funciona con tipos anónimos:

```
var a1 = new { A = 1, B = 2, C = 3, D = 4, E = 5 };
var a2 = a1 with { E = 10 };
Console.WriteLine (a2);      // { A = 1, B = 2, C = 3, D = 4, E = 10 }
```

Nueva sintaxis de desestructuración

C# 7 introdujo la sintaxis de desestructuración para tuplas (o cualquier tipo con un método Deconstruct). C# 10 lleva esta sintaxis más lejos, permitiéndote combinar asignación y declaración en la misma desestructuración:

```
var point = (3, 4);
double x = 0;
(x, double y) = point;
```

Inicializadores de campo y constructores sin parámetros en estructuras

A partir de C# 10, puedes incluir inicializadores de campo y constructores sin parámetros en estructuras (ver "Estructuras" en la página 129). Estos se ejecutan solo cuando el constructor es llamado explícitamente y pueden ser fácilmente omitidos, por ejemplo, mediante la palabra clave default. Esta característica fue introducida principalmente para beneficiar a las estructuras de tipo registro.

Estructuras de tipo registro

Las estructuras de tipo registro se introdujeron por primera vez en C# 9, donde actuaban como una clase mejorada compilada. En C# 10, los registros también pueden ser estructuras:

```
record struct Point (int X, int Y);
```

Las reglas son similares en otros aspectos: las estructuras de tipo registro tienen muchas de las mismas características que las estructuras de clase (ver "Registros" en la página 211). Una excepción es que las propiedades generadas por el compilador en las estructuras de tipo registro son de escritura, a menos que antepongas la palabra clave readonly en la declaración del registro.

Mejoras en las expresiones lambda

La sintaxis alrededor de las expresiones lambda se ha mejorado de varias formas. Primero, se permite la tipificación implícita (var):

```
var greeter = () => "Hello, world";
```

El tipo implícito para una expresión lambda es un delegado Action o Func, por lo que greeter, en este caso, es de tipo Func<string>. Debes especificar explícitamente los tipos de los parámetros:

```
var square = (int x) => x * x;
```

En segundo lugar, una expresión lambda puede especificar un tipo de retorno:

```
var sqr = int (int x) => x;
```

Esto se hace principalmente para mejorar el rendimiento del compilador con expresiones lambda anidadas complejas.

En tercer lugar, puedes pasar una expresión lambda como argumento a un parámetro de método de tipo object, Delegate o Expression:

```
M1 (() => "test");   // Implicitly typed to Func<string>
M2 (() => "test");   // Implicitly typed to Func<string>
M3 (() => "test");   // Implicitly typed to Expression<Func<string>>

void M1 (object x) {}
void M2 (Delegate x) {}
void M3 (Expression x) {}
```

Por último, puedes aplicar atributos al método de destino generado por compilación de una expresión lambda (así como a sus parámetros y valor de retorno):

```
Action a = [Description("test")] () => { };
```

Consulta "Aplicar atributos a expresiones lambda (C# 10)" en la página 228 para más detalles.

Patrones de propiedades anidadas

La siguiente sintaxis simplificada es válida en C# 10 para la coincidencia de patrones de propiedades anidadas (consulta "Patrones de propiedades" en la página 225):

```
var obj = new Uri ("https://www.linqpad.net");
if (obj is Uri { Scheme.Length: 5 }) ...
```

Esto es equivalente a:

```
if (obj is Uri { Scheme: { Length: 5 }}) ...
```

CallerArgumentExpression

Un parámetro de método al que se le aplica el atributo [CallerArgumentExpression] captura una expresión de argumento del sitio de llamada:

```
Print (Math.PI * 2);

void Print (double number,
            [CallerArgumentExpression("number")] string expr = null)
  => Console.WriteLine (expr);

// Output: Math.PI * 2
```

Esta característica está destinada principalmente a bibliotecas de validación y aserción (ver "Caller-ArgumentExpression (C# 10)" en la página 231).

Otras nuevas características

La directiva #line se ha mejorado en C# 10 para permitir especificar una columna y un rango.

Las cadenas interpoladas en C# 10 pueden ser constantes, siempre y cuando los valores interpolados también sean constantes.

Los registros pueden sellar el método ToString() en C# 10.

El análisis de asignación definitiva de C# se ha mejorado para que expresiones como la siguiente funcionen:

```
if (foo?.TryParse ("123", out var number) ?? false)
  Console.WriteLine (number);
```

(Previo a C# 10, el compilador generaría un error: "Uso de la variable local no asignada 'number'").

Novedades en C# 9.0

C# 9.0 se lanzó con Visual Studio 2019 y se utiliza al apuntar a .NET 5.

Declaraciones en el nivel superior

Con las declaraciones en el nivel superior (ver "Declaraciones en el nivel superior"), puedes escribir un programa sin la estructura de un método Main y una clase Program:

```
using System;
Console.WriteLine ("Hello, world");
```

Las declaraciones en el nivel superior pueden incluir métodos (que actúan como métodos locales). También puedes acceder a los argumentos de la línea de comandos a través de la variable "mágica" args y devolver un valor al llamador. Las declaraciones en el nivel superior pueden ir seguidas de declaraciones de tipo y espacio de nombres.

Inicializadores de solo lectura

Un inicializador de solo lectura (ver "Inicializadores de solo lectura") en una declaración de propiedad utiliza la palabra clave init en lugar de la palabra clave set:

```
class Foo { public int ID { get; init; } }
```

Esta característica se comporta como una propiedad de solo lectura, excepto que también se puede establecer a través de un inicializador de objetos:

```
var foo = new Foo { ID = 123 };
```

Esto permite crear tipos inmutables (de solo lectura) que se pueden llenar mediante un inicializador de objetos en lugar de un constructor, y ayuda a evitar el anti-patrón de constructores que aceptan un gran número de parámetros opcionales. Los inicializadores de solo lectura también permiten la mutación no destructiva cuando se utilizan en registros.

Registros

Un registro (ver "Registros" en la página 211) es un tipo especial de clase diseñado para funcionar bien con datos inmutables. Su característica más especial es que admite mutación no destructiva mediante una nueva palabra clave (with):

```
Point p1 = new Point (2, 3);
Point p2 = p1 with { Y = 4 };   // p2 is a copy of p1, but with Y set to 4
Console.WriteLine (p2);         // Point { X = 2, Y = 4 }

record Point
{
  public Point (double x, double y) => (X, Y) = (x, y);

  public double X { get; init; }
  public double Y { get; init; }
}
```

En casos simples, un registro también puede eliminar el código repetitivo de definir propiedades y escribir un constructor y un deconstructor. Podemos reemplazar nuestra definición de registro Point con lo siguiente, sin perder funcionalidad:

```
record Point (double X, double Y);
```

Al igual que las tuplas, los registros exhiben igualdad estructural de forma predeterminada. Los registros pueden heredar de otros registros e incluir las mismas construcciones que pueden incluir las clases. El compilador implementa los registros como clases en tiempo de ejecución.

Mejoras en la coincidencia de patrones

El patrón relacional (ver "Patrones" en la página 222) permite que los operadores <, >, <= y >= aparezcan en patrones:

```
string GetWeightCategory (decimal bmi) => bmi switch {
  < 18.5m => "underweight",
  < 25m => "normal",
  < 30m => "overweight",
  _ => "obese" };
```

Con los combinadores de patrones, se pueden combinar patrones mediante tres nuevas palabras clave (and, or y not):

```
bool IsVowel (char c) => c is 'a' or 'e' or 'i' or 'o' or 'u';


bool IsLetter (char c) => c is >= 'a' and <= 'z'
                           or >= 'A' and <= 'Z';
```

Al igual que los operadores && y ||, el operador and tiene mayor precedencia que or. Puedes anular esto con el uso de paréntesis.

El combinador not se puede utilizar con el patrón de tipo para comprobar si un objeto es (o no es) de un tipo específico:

```
if (obj is not string) ...
```

Expresiones de nuevo tipo con tipo objetivo

Cuando se construye un objeto, C# 9 te permite omitir el nombre del tipo cuando el compilador puede inferirlo de manera unívoca:

```
System.Text.StringBuilder sb1 = new();
System.Text.StringBuilder sb2 = new ("Test");
```

Esto es particularmente útil cuando la declaración e inicialización de la variable se encuentran en diferentes partes de tu código:

```
class Foo
{
  System.Text.StringBuilder sb;
  public Foo (string initialValue) => sb = new (initialValue);
}
```

Y en el siguiente escenario:

```
MyMethod (new ("test"));
void MyMethod (System.Text.StringBuilder sb) { ... }
```

Ver "Expresiones de nuevo tipo con tipo objetivo" en la página 69 para obtener más información.

Mejoras de interoperabilidad

C# 9 introdujo punteros a funciones (ver "Punteros a funciones" en la página 248 y "Retornos de llamada con punteros a funciones" en la página 967). Su objetivo principal es permitir que el código no administrado llame a métodos estáticos en C# sin la sobrecarga de una instancia de delegado, con la capacidad de omitir la capa de P/Invoke cuando los argumentos y tipos de retorno se pueden representar de forma idéntica en ambos lados.

C# 9 también introdujo los tipos de entero nativos nint y nuint (ver "Enteros de tamaño nativo" en la página 246), que se mapean en tiempo de ejecución a System.IntPtr y System.UIntPtr. En tiempo de compilación, se comportan como tipos numéricos con soporte para operaciones aritméticas.

Otras nuevas características

Además, C# 9 ahora te permite hacer lo siguiente:

• Sobrescribir un método o propiedad de solo lectura de manera que devuelva un tipo más derivado (ver "Tipos de retorno covariantes" en la página 120)

Aplicar atributos a funciones locales (ver "Atributos" en la página 227)

• Aplicar la palabra clave static a expresiones lambda o funciones locales para asegurarte de que no capturas accidentalmente variables locales o de instancia (ver "Expresiones lambda estáticas" en la página 177).

• Hacer que cualquier tipo funcione con la instrucción foreach, escribiendo un método de extensión GetEnumerator.

• Definir un método de inicialización de módulo que se ejecuta una vez cuando se carga por primera vez un ensamblado, aplicando el atributo [ModuleInitializer] a un método (void estático sin parámetros).

• Utilizar un "descarte" (símbolo de guion bajo) como argumento de una expresión lambda.

• Escribir métodos parciales extendidos que sean obligatorios de implementar, lo que permite escenarios como los nuevos generadores de código fuente de Roslyn (ver "Métodos parciales extendidos" en la página 114).

• Aplicar un atributo a métodos, tipos o módulos para evitar que las variables locales sean inicializadas por el tiempo de ejecución (ver "[SkipLocalsInit]" en la página 248).

Novedades en C# 8.0

C# 8.0 se lanzó por primera vez con Visual Studio 2019 y todavía se utiliza hoy en día cuando se apunta a .NET Core 3 o .NET Standard 2.1.

Índices y rangos

Los índices y rangos simplifican el trabajo con elementos o partes de un arreglo (o los tipos de bajo nivel Span<T> y ReadOnlySpan<T>).

Los índices te permiten referirte a elementos en relación con el final de un arreglo utilizando el operador ^. ^1 se refiere al último elemento, ^2 se refiere al penúltimo elemento, y así sucesivamente:

```
char[] vowels = new char[] {'a','e','i','o','u'};
char lastElement  = vowels [^1];   // 'u'
char secondToLast = vowels [^2];   // 'o'
```

Los rangos te permiten "cortar" un arreglo utilizando el operador ..:

```
char[] firstTwo =  vowels [..2];    // 'a', 'e'
char[] lastThree = vowels [2..];    // 'i', 'o', 'u'
char[] middleOne = vowels [2..3]    // 'i'
char[] lastTwo =   vowels [^2..];   // 'o', 'u'
```

C# implementa índices y rangos con la ayuda de los tipos Index y Range:

```
Index last = ^1;
Range firstTwoRange = 0..2;
char[] firstTwo = vowels [firstTwoRange];   // 'a', 'e'
```

Puedes admitir índices y rangos en tus propias clases definiendo un indexador con un tipo de parámetro Index o Range:

```
class Sentence
{
  string[] words = "The quick brown fox".Split();

  public string this   [Index index] => words [index];
  public string[] this [Range range] => words [range];
}
```

Para obtener más información, consulta "Índices" en la página 56.

Asignación de fusión nula:

El operador ??= asigna una variable solo si es nula. En lugar de:

```
if (s == null) s = "Hello, world";
```

puedes escribir esto:

```
s ??= "Hello, world";
```

Declaraciones de uso:

Si omites los corchetes y el bloque de instrucciones que sigue a una declaración de uso, se convierte en una declaración de uso. El recurso se elimina cuando la ejecución sale del bloque de instrucciones que lo engloba:

```
if (File.Exists ("file.txt"))
{
  using var reader = File.OpenText ("file.txt");
  Console.WriteLine (reader.ReadLine());
  ...
}
```

En este caso, el lector (reader) se eliminará cuando la ejecución salga del bloque if.

Miembros de solo lectura:

C# 8 te permite aplicar el modificador readonly a las funciones de una estructura (struct), lo que garantiza que si la función intenta modificar algún campo, se genere un error en tiempo de compilación:

```
struct Point
{
  public int X, Y;
  public readonly void ResetX() => X = 0;  // Error!
}
```

Si una función de solo lectura llama a una función que no es de solo lectura, el compilador genera una advertencia (y copia defensivamente la estructura para evitar la posibilidad de una mutación).

Métodos locales estáticos:

Al agregar el modificador static a un método local, se evita que este tenga acceso a las variables locales y parámetros del método que lo contiene. Esto ayuda a reducir el acoplamiento y permite que el método local declare variables a su gusto, sin riesgo de colisionar con las variables del método que lo contiene.

Miembros predeterminados de interfaces:

C# 8 te permite agregar una implementación predeterminada a un miembro de una interfaz, lo que lo hace opcional de implementar:

```
interface ILogger
{
  void Log (string text) => Console.WriteLine (text);
}
```

Esto significa que puedes agregar un miembro a una interfaz sin romper las implementaciones existentes. Las implementaciones predeterminadas deben ser llamadas explícitamente a través de la interfaz:

```
((ILogger)new Logger()).Log ("message");
```

Las interfaces también pueden definir miembros estáticos (incluyendo campos), los cuales pueden ser accedidos desde el código dentro de las implementaciones predeterminadas:

```
interface ILogger
{
  void Log (string text) => Console.WriteLine (Prefix + text);
  static string Prefix = "";
}
```

O desde fuera de la interfaz, a menos que esté restringido mediante un modificador de accesibilidad en el miembro estático de la interfaz (como private, protected o internal):

```
ILogger.Prefix = "File log: ";
```

Los campos de instancia están prohibidos. Para obtener más detalles, consulta "Miembros predeterminados de interfaces" en la página 139.

Expresiones de switch:

A partir de C# 8, puedes utilizar switch en el contexto de una expresión:

```
string cardName = cardNumber switch    // assuming cardNumber is an int
{
  13 => "King",
  12 => "Queen",
  11 => "Jack",
  _ => "Pip card"   // equivalent to 'default'
};
```

Para obtener más ejemplos, consulta "Expresiones de switch" en la página 82.

Patrones de tuplas, posicionales y de propiedades

C# 8 introduce tres nuevos patrones, principalmente para beneficiar a las sentencias/expresiones de switch (consulta "Patrones" en la página 222). Los patrones de tuplas te permiten realizar un switch en múltiples valores:

```
int cardNumber = 12; string suite = "spades";
string cardName = (cardNumber, suite) switch
{
  (13, "spades") => "King of spades",
  (13, "clubs") => "King of clubs",
  ...
};
```

Los patrones posicionales permiten una sintaxis similar para objetos que exponen un desconstructor, y los patrones de propiedades te permiten hacer coincidir propiedades de un objeto. Puedes utilizar todos estos patrones tanto en sentencias de switch como con el operador is. El siguiente ejemplo utiliza un patrón de propiedad para comprobar si obj es una cadena con una longitud de 4:

```
if (obj is string { Length:4 }) ...
```

Tipos de referencia nulos

Mientras que los tipos de valor nulos brindan la posibilidad de tener nulabilidad en tipos de valor, los tipos de referencia nulos hacen lo contrario y brindan (en cierto grado) no nulabilidad a los tipos de referencia, con el propósito de ayudar a evitar las excepciones de referencia nula (NullReferenceException). Los tipos de referencia nulos introducen un nivel de seguridad que es aplicado puramente por el compilador en forma de advertencias o errores cuando detecta código que está en riesgo de generar una NullReferenceException.

Los tipos de referencia nulos se pueden habilitar a nivel de proyecto (mediante el elemento Nullable en el archivo de proyecto .csproj) o en el código (mediante la directiva #nullable). Una vez habilitados, el compilador establece la no nulabilidad como predeterminada: si deseas que un tipo de referencia acepte valores nulos, debes aplicar el sufijo ? para indicar un tipo de referencia nulo:

```
#nullable enable    // Enable nullable reference types from this point on

string s1 = null;   // Generates a compiler warning! (s1 is non-nullable)
string? s2 = null;  // OK: s2 is nullable reference type
```

Los campos no inicializados también generan una advertencia (si el tipo no está marcado como nulo), al igual que al desreferenciar un tipo de referencia nulo si el compilador considera que podría producirse una NullReferenceException:

```
void Foo (string? s) => Console.Write (s.Length);  // Warning (.Length)
```

Para eliminar la advertencia, puedes utilizar el operador de confianza nula (!):

```
void Foo (string? s) => Console.Write (s!.Length);
```

Para obtener una discusión completa, consulta "Tipos de referencia nulos" en la página 200.

Flujos asíncronos

Antes de C# 8, podías utilizar yield return para escribir un iterador o await para escribir una función asíncrona. Sin embargo, no podías hacer ambas cosas y escribir un iterador que esperara, devolviendo elementos de manera asíncrona. C# 8 soluciona esto mediante la introducción de flujos asíncronos:

```
async IAsyncEnumerable<int> RangeAsync (
  int start, int count, int delay)
{
  for (int i = start; i < start + count; i++)
  {
    await Task.Delay (delay);
    yield return i;
  }
}
```

La sentencia await foreach consume un flujo asíncrono:

```
await foreach (var number in RangeAsync (0, 10, 100))
  Console.WriteLine (number);
```

Para obtener más información, consulta "Flujos Asíncronos" en la página 650.

Novedades en C# 7.x

C# 7.x se lanzó por primera vez con Visual Studio 2017. C# 7.3 todavía se utiliza hoy en día en Visual Studio 2019 cuando se apunta a .NET Core 2, .NET Framework 4.6 a 4.8 o .NET Standard 2.0.

C# 7.3

C# 7.3 introdujo mejoras menores a características existentes, como habilitar el uso de operadores de igualdad con tuplas, mejorar la resolución de sobrecargas y ofrecer la capacidad de aplicar atributos a los campos subyacentes de propiedades automáticas:

```
[field:NonSerialized]
public int MyProperty { get; set; }
```

C# 7.3 también se basó en las características avanzadas de programación de baja asignación de C# 7.2, con la capacidad de volver a asignar referencias locales, no requerir anclaje al indexar campos fijos y admitir inicializadores de campo con stackalloc:

```
int* pointer  = stackalloc int[] {1, 2, 3};
Span<int> arr = stackalloc []    {1, 2, 3};
```

Observa que la memoria asignada en el stack puede asignarse directamente a un Span<T>. Describimos los spans, y por qué los usarías, en el Capítulo 23.

C# 7.2

C# 7.2 agregó un nuevo modificador private protected (la intersección de internal y protected), la capacidad de seguir argumentos con nombre con argumentos posicionales al llamar a métodos y estructuras de solo lectura. Una struct de solo lectura (readonly struct) asegura que todos los campos sean de solo lectura, lo cual ayuda a declarar la intención y permite que el compilador tenga más libertad de optimización:

```
readonly struct Point
{
  public readonly int X, Y;   // X and Y must be readonly
}
```

C# 7.2 también agregó características especializadas para ayudar con la micro-optimización y la programación de baja asignación de memoria: consulta "El modificador in" en la página 64, "Referencias locales" en la página 67, "Retornos por referencia" en la página 68 y "Structs por referencia" en la página 131.

C# 7.1

A partir de C# 7.1, puedes omitir el tipo al usar la palabra clave "default", si el tipo puede ser inferido:

```
decimal number = default;   // number is decimal
```

C# 7.1 también relajó las reglas para las sentencias switch (permitiendo realizar coincidencias de patrones en parámetros de tipo genérico), permitió que el método Main de un programa sea asíncrono y permitió inferir los nombres de los elementos de una tupla:

```
var now = DateTime.Now;
var tuple = (now.Hour, now.Minute, now.Second);
```

Mejoras en literales numéricos

En C# 7, los literales numéricos pueden incluir guiones bajos para mejorar la legibilidad. Estos se llaman separadores de dígitos y son ignorados por el compilador:

```
int million = 1_000_000;
```

Los literales binarios pueden ser especificados con el prefijo "0b":

```
var b = 0b1010_1011_1100_1101_1110_1111;
```

Variables de salida y descartes

C# 7 facilita llamar a métodos que contienen parámetros de salida (out parameters). Primero, ahora puedes declarar variables de salida al vuelo (ver "Variables de salida y descartes" en la página 64):

```
bool successful = int.TryParse ("123", out int result);
Console.WriteLine (result);
```

Y al llamar a un método con múltiples parámetros de salida, puedes descartar aquellos en los que no estás interesado utilizando el carácter guion bajo:

```
SomeBigMethod (out _, out _, out _, out int x, out _, out _, out _);
Console.WriteLine (x);
```

Patrones de tipo y variables de patrón

También puedes introducir variables al vuelo con el operador "is". Estas se llaman variables de patrón (pattern variables) (ver "Introducción a una variable de patrón" en la página 118):

```
void Foo (object x)
{
  if (x is string s)
    Console.WriteLine (s.Length);
}
```

La sentencia switch también admite patrones de tipo, por lo que puedes hacer switch en el tipo además de en constantes (consulta "Switching on types" en la página 81). Puedes especificar condiciones con una cláusula when y también hacer switch en el valor nulo:

```
switch (x)
{
  case int i:
    Console.WriteLine ("It's an int!");
    break;
  case string s:
    Console.WriteLine (s.Length);    // We can use the s variable
    break;
  case bool b when b == true:        // Matches only when b is true
    Console.WriteLine ("True");
    break;
  case null:
    Console.WriteLine ("Nothing");
    break;
}
```

Métodos locales

Un método local es un método declarado dentro de otra función (consulta "Local methods" en la página 98):

```
void WriteCubes()
{
  Console.WriteLine (Cube (3));
  Console.WriteLine (Cube (4));
  Console.WriteLine (Cube (5));

  int Cube (int value) => value * value * value;
}
```

Los métodos locales son visibles solo para la función que los contiene y pueden capturar variables locales de la misma manera que lo hacen las expresiones lambda.

Más miembros expresados en forma de expresión

C# 6 introdujo la sintaxis de "flecha gorda" (fat-arrow) expresada en forma de expresión para métodos, propiedades de solo lectura, operadores e indexadores. C# 7 extiende esto a constructores, propiedades de lectura/escritura y finalizadores:

```
public class Person
{
  string name;

  public Person (string name) => Name = name;
```

```
  public string Name
  {
    get => name;
    set => name = value ?? "";
  }

  ~Person () => Console.WriteLine ("finalize");
}
```

Desconstructores

C# 7 introduce el patrón de deconstructor (consulta "Deconstructors" en la página 102). Mientras que un constructor típicamente toma un conjunto de valores (como parámetros) y los asigna a campos, un deconstructor hace lo contrario y asigna los campos de nuevo a un conjunto de variables. Podríamos escribir un deconstructor para la clase Person del ejemplo anterior de la siguiente manera (sin tener en cuenta el manejo de excepciones):

```
public void Deconstruct (out string firstName, out string lastName)
{
  int spacePos = name.IndexOf (' ');
  firstName = name.Substring (0, spacePos);
  lastName = name.Substring (spacePos + 1);
}
```

Los deconstructores se llaman con la siguiente sintaxis especial:

```
var joe = new Person ("Joe Bloggs");
var (first, last) = joe;          // Deconstruction
Console.WriteLine (first);        // Joe
Console.WriteLine (last);         // Bloggs
```

Tuplas

Quizás la mejora más notable en C# 7 es el soporte explícito para las tuplas (consulta "Tuples" en la página 207). Las tuplas proporcionan una forma sencilla de almacenar un conjunto de valores relacionados:

```
var bob = ("Bob", 23);
Console.WriteLine (bob.Item1);   // Bob
Console.WriteLine (bob.Item2);   // 23
```

Las nuevas tuplas de C# son azúcar sintáctica para usar las estructuras genéricas System.ValueTuple<...>. Pero gracias a la magia del compilador, los elementos de la tupla pueden tener nombres:

```
var tuple = (name:"Bob", age:23);
Console.WriteLine (tuple.name);      // Bob
Console.WriteLine (tuple.age);       // 23
```

Con las tuplas, las funciones pueden devolver múltiples valores sin recurrir a parámetros de salida o a un tipo adicional.

```
static (int row, int column) GetFilePosition() => (3, 10);

static void Main()
{
  var pos = GetFilePosition();


    Console.WriteLine (pos.row);       // 3
    Console.WriteLine (pos.column);    // 10
 }
```

Las tuplas admiten implícitamente el patrón de desestructuración, por lo que puedes desestructurarlas fácilmente en variables individuales:

```
static void Main()
{
  (int row, int column) = GetFilePosition();   // Creates 2 local variables
  Console.WriteLine (row);       // 3
  Console.WriteLine (column);    // 10
}
```

Expresiones de lanzamiento de excepciones

Antes de C# 7, throw siempre era una declaración. Ahora también puede aparecer como una expresión en funciones de cuerpo de expresión:

```
public string Foo() => throw new NotImplementedException();
```

Una expresión de lanzamiento también puede aparecer en una expresión condicional ternaria:

```
string Capitalize (string value) =>
  value == null ? throw new ArgumentException ("value") :
  value == "" ? "" :
  char.ToUpper (value[0]) + value.Substring (1);
```

Novedades en C# 6.0

C# 6.0, que se lanzó con Visual Studio 2015, presenta un compilador de nueva generación completamente escrito en C#. Conocido como proyecto "Roslyn", el nuevo compilador expone toda la cadena de compilación a través de bibliotecas, lo que te permite realizar análisis de código en código fuente arbitrario. El compilador en sí es de código abierto y el código fuente está disponible en github.com/dotnet/roslyn.

Además, C# 6.0 incluye varias mejoras menores pero significativas, orientadas principalmente a reducir el desorden de código.

El operador de nulabilidad condicional ("operador Elvis", consulta "Null Operators" en la página 74) evita tener que verificar explícitamente si es nulo antes de llamar a un método o acceder a un miembro de tipo. En el siguiente ejemplo, result se evalúa como null en lugar de lanzar una NullReferenceException:

```
System.Text.StringBuilder sb = null;
string result = sb?.ToString();      // result is null
```

Las funciones de cuerpo de expresión (ver "Métodos" en la página 98) permiten que los métodos, propiedades, operadores e indexadores que constan de una sola expresión se escriban de forma más concisa, en el estilo de una expresión lambda:

```
public int TimesTwo (int x) => x * 2;
public string SomeProperty => "Property value";
```

Inicializadores de propiedades (Capítulo 3) te permiten asignar un valor inicial a una propiedad automática:

```
public DateTime TimeCreated { get; set; } = DateTime.Now;
```

Las propiedades inicializadas también pueden ser de solo lectura:

```
public DateTime TimeCreated { get; } = DateTime.Now;
```

Las propiedades de solo lectura también se pueden establecer en el constructor, lo que facilita la creación de tipos inmutables (de solo lectura).

Inicializadores de índices (Capítulo 4) permiten la inicialización en un solo paso de cualquier tipo que exponga un indexador:

```
var dict = new Dictionary<int,string>()
{
  [3] = "three",
  [10] = "ten"
};
```

Interpolación de cadenas (ver "Tipo String" en la página 52) ofrece una alternativa concisa a string.Format:

```
string s = $"It is {DateTime.Now.DayOfWeek} today";
```

Los filtros de excepción (ver "Sentencias try y excepciones" en la página 181) te permiten aplicar una condición a un bloque catch:

```
string html;
try
{
  html = await new HttpClient().GetStringAsync ("http://asef");
}
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
  ...
}
```

La directiva using static (ver "Espacios de nombres" en la página 87) te permite importar todos los miembros estáticos de un tipo para que puedas usar esos miembros sin calificarlos:

```
using static System.Console;
...
WriteLine ("Hello, world");  // WriteLine instead of Console.WriteLine
```

El operador nameof (Capítulo 3) devuelve el nombre de una variable, tipo u otro símbolo como una cadena. Esto evita romper el código cuando renombras un símbolo en Visual Studio:

```
int capacity = 123;
string x = nameof (capacity);   // x is "capacity"
string y = nameof (Uri.Host);   // y is "Host"
```

Y por último, ahora se permite usar await dentro de los bloques catch y finally.

Lo nuevo en C# 5.0

La gran característica nueva de C# 5.0 fue el soporte para funciones asíncronas a través de dos nuevas palabras clave, async y await. Las funciones asíncronas permiten continuaciones asíncronas, lo que facilita la escritura de aplicaciones de cliente rico responsivas y seguras en términos de hilos. También facilitan la escritura de aplicaciones de E/S intensiva altamente concurrentes y eficientes que no ocupan un recurso de hilo por operación. Cubrimos las funciones asíncronas en detalle en el Capítulo 14.

Lo nuevo en C# 4.0

C# 4.0 introdujo cuatro mejoras principales:

Enlace dinámico (Capítulos 4 y 19) aplaza el enlace, el proceso de resolución de tipos y miembros, desde el tiempo de compilación hasta el tiempo de ejecución, y es útil en escenarios que de otra manera requerirían código de reflexión complicado. El enlace dinámico también es útil al interoperar con lenguajes dinámicos y componentes COM.

Parámetros opcionales (Capítulo 2) permiten que las funciones especifiquen valores predeterminados para los parámetros, de modo que los llamadores puedan omitir argumentos, y los argumentos nombrados permiten que el llamador de una función identifique un argumento por nombre en lugar de por posición.

Se relajaron las reglas de covarianza y contravarianza en C# 4.0 (Capítulos 3 y 4), de modo que los parámetros de tipo en interfaces genéricas y delegados genéricos pueden marcarse como covariantes o contravariantes, lo que permite conversiones de tipo más naturales.

La interoperabilidad con COM (Capítulo 24) se mejoró en C# 4.0 de tres formas. Primero, se pueden pasar argumentos por referencia sin la palabra clave ref (particularmente útil en conjunto con parámetros opcionales). Segundo, los ensamblados que contienen tipos de interoperabilidad con COM se pueden vincular en lugar de referenciar. Los tipos de interoperabilidad vinculados admiten equivalencia de tipos, evitando la necesidad de ensamblados primarios de interoperabilidad y poniendo fin a los problemas de versionado y implementación. Tercero, las funciones que devuelven tipos Variant de COM a partir de tipos de interoperabilidad vinculados se asignan a dynamic en lugar de object, eliminando la necesidad de conversiones explícitas.

Lo nuevo en C# 3.0

Las características agregadas a C# 3.0 se centraron principalmente en las capacidades de Language-Integrated Query (LINQ). LINQ permite escribir consultas directamente dentro de un programa en C# y verificarlas estáticamente para detectar errores, y puede realizar consultas tanto en colecciones locales (como listas o documentos XML) como en fuentes de datos remotas (como una base de datos). Las características de C# 3.0 agregadas para admitir LINQ incluyen variables locales de tipo implícito, tipos anónimos, inicializadores de objetos, expresiones lambda, métodos de extensión, expresiones de consulta y árboles de expresión.

Las variables locales de tipo implícito (palabra clave var, Capítulo 2) te permiten omitir el tipo de variable en una declaración, permitiendo que el compilador lo infiera. Esto reduce el desorden y también permite tipos anónimos (Capítulo 4), que son clases simples creadas sobre la marcha y que se usan comúnmente en el resultado final de las consultas LINQ. También puedes usar la tipificación implícita en arreglos (Capítulo 2).

Los inicializadores de objetos (Capítulo 3) simplifican la construcción de objetos al permitirte establecer propiedades en línea después de la llamada al constructor. Los inicializadores de objetos funcionan tanto con tipos con nombre como con tipos anónimos.

Las expresiones lambda (Capítulo 4) son funciones en miniatura creadas por el compilador sobre la marcha; son particularmente útiles en consultas LINQ "fluentes" (Capítulo 8).

Los métodos de extensión (Capítulo 4) amplían un tipo existente con nuevos métodos (sin alterar la definición del tipo), lo que hace que los métodos estáticos se sientan como métodos de instancia. Los operadores de consulta de LINQ se implementan como métodos de extensión.

Las expresiones de consulta (Capítulo 8) proporcionan una sintaxis de nivel superior para escribir consultas LINQ que pueden ser considerablemente más simples cuando se trabaja con múltiples secuencias o variables de rango.

Los árboles de expresión (Capítulo 8) son modelos de objetos de código en miniatura que describen expresiones lambda asignadas al tipo especial Expression<TDelegate>. Los árboles de expresión hacen posible que las consultas LINQ se ejecuten de forma remota (por ejemplo, en un servidor de base de datos) porque pueden ser inspeccionados y traducidos en tiempo de ejecución (por ejemplo, en una declaración SQL).

C# 3.0 también agregó propiedades automáticas y métodos parciales.

Las propiedades automáticas (Capítulo 3) reducen el trabajo de escribir propiedades que simplemente obtienen/establecen un campo de respaldo privado al hacer que el compilador haga ese trabajo automáticamente.

Los métodos parciales (Capítulo 3) permiten que una clase parcial generada automáticamente proporcione puntos de enganche personalizables para la autoría manual que "desaparecen" si no se utilizan.

Qué hay de nuevo en C# 2.0

Las características principales en C# 2 fueron los genéricos (Capítulo 3), los tipos de valor anulables (Capítulo 4), los iteradores (Capítulo 4) y los métodos anónimos (el predecesor de las expresiones lambda). Estas características allanaron el camino para la introducción de LINQ en C# 3.

C# 2 también agregó soporte para clases parciales y clases estáticas, y una serie de características menores y diversas como el calificador de alias de espacio de nombres, ensamblados amigos y búferes de tamaño fijo.

La introducción de genéricos requirió un nuevo CLR (CLR 2.0), porque los genéricos mantienen plena fidelidad de tipo en tiempo de ejecución.

2 Fundamentos del lenguaje C#

En este capítulo, presentamos los conceptos básicos del lenguaje C#.



La mayoría de los fragmentos de código de este libro están disponibles como muestras interactivas en LINQPad. Trabajar con estas muestras junto con el libro acelera el aprendizaje, ya que puedes editar las muestras y ver los resultados instantáneamente sin necesidad de configurar proyectos y soluciones en Visual Studio.

Para descargar las muestras, en LINQPad, haz clic en la pestaña "Samples" y luego en "Download more samples". LINQPad es gratuito, puedes obtenerlo en http://www.linqpad.net.

Un Primer Programa en C#

A continuación se muestra un programa que multiplica 12 por 30 e imprime el resultado, 360, en la pantalla. Las dos barras diagonales indican que el resto de la línea es un comentario:

```
int x = 12 * 30;                    // Statement 1
System.Console.WriteLine (x);       // Statement 2
```

Nuestro programa consta de dos declaraciones. Las declaraciones en C# se ejecutan secuencialmente y se terminan con un punto y coma. La primera declaración calcula la expresión 12 * 30 y almacena el resultado en una variable llamada x, cuyo tipo es un entero de 32 bits (int). La segunda declaración llama al método WriteLine en una clase llamada Console, que está definida en un espacio de nombres llamado System. Esto imprime la variable x en una ventana de texto en la pantalla.

Un método realiza una función; una clase agrupa miembros de función y miembros de datos para formar un bloque de construcción orientado a objetos. La clase Console agrupa miembros que manejan la funcionalidad de entrada/salida (E/S) de la línea de comandos, como el método WriteLine. Una clase es un tipo de dato, que examinamos en "Fundamentos de los Tipos" en la página 30.

En el nivel más externo, los tipos se organizan en espacios de nombres (namespaces). Muchos tipos comúnmente utilizados, incluida la clase Console, se encuentran en el espacio de nombres System. Las bibliotecas .NET están organizadas en espacios de nombres anidados. Por ejemplo, el espacio de nombres System.Text contiene tipos para manejar texto, y System.IO contiene tipos para entrada/salida.

Calificar la clase Console con el espacio de nombres System en cada uso añade desorden. La directiva using te permite evitar este desorden importando un espacio de nombres:

```
using System;               // Import the System namespace

int x = 12 * 30;
Console.WriteLine (x);      // No need to specify System.
```

Una forma básica de reutilización de código es escribir funciones de nivel superior que llaman a funciones de nivel inferior. Podemos refactorizar nuestro programa con un método reutilizable llamado FeetToInches que multiplica un entero por 12, de la siguiente manera:

```
using System;

Console.WriteLine (FeetToInches (30));      // 360
Console.WriteLine (FeetToInches (100));     // 1200

int FeetToInches (int feet)
{
  int inches = feet * 12;
  return inches;
}
```

Nuestro método contiene una serie de declaraciones rodeadas por un par de llaves. Esto se llama bloque de declaraciones (statement block).

Un método puede recibir datos de entrada del llamador al especificar parámetros y devolver datos de salida al llamador al especificar un tipo de retorno. Nuestro método FeetToInches tiene un parámetro para introducir los pies y un tipo de retorno para devolver las pulgadas:

```
int FeetToInches (int feet)
...
```

Los literales 30 y 100 son los argumentos pasados al método FeetToInches.

Si un método no recibe entrada, se utilizan paréntesis vacíos. Si no devuelve nada, se utiliza la palabra clave void:

```
using System;
SayHello();

void SayHello()
{
  Console.WriteLine ("Hello, world");
}
```

Los métodos son uno de varios tipos de funciones en C#. Otro tipo de función que utilizamos en nuestro programa de ejemplo fue el operador * que realiza multiplicación. También existen constructores, propiedades, eventos, indexadores y finalizadores.

Compilación

El compilador de C# compila el código fuente (un conjunto de archivos con la extensión .cs) en un ensamblado. Un ensamblado es la unidad de empaquetado e implementación en .NET. Un ensamblado puede ser una aplicación o una biblioteca. Una aplicación normal de consola o Windows tiene un punto de entrada, mientras que una biblioteca no lo tiene. El propósito de una biblioteca es ser llamada (referenciada) por una aplicación u otras bibliotecas. .NET en sí mismo es un conjunto de bibliotecas (además de un entorno de ejecución).

Cada uno de los programas en la sección anterior comenzó directamente con una serie de declaraciones (llamadas declaraciones de nivel superior). La presencia de declaraciones de nivel superior crea implícitamente un punto de entrada para una aplicación de consola o Windows. (Sin declaraciones de nivel superior, un método Main denota el punto de entrada de una aplicación. Consulta "Tipos personalizados" en la página 31 para más detalles).



A diferencia de .NET Framework, los ensamblados de .NET 6 nunca tienen una extensión .exe. El archivo .exe que ves después de compilar una aplicación de .NET 6 es un cargador nativo específico de la plataforma responsable de iniciar el ensamblado .dll de tu aplicación.

.NET 6 también te permite crear una implementación autocontenida que incluye el cargador, tus ensamblados y el tiempo de ejecución de .NET, todo en un solo archivo .exe.

La herramienta dotnet (dotnet.exe en Windows) te ayuda a administrar el código fuente y los binarios de .NET desde la línea de comandos. Puedes utilizarlo tanto para compilar como para ejecutar tu programa, como alternativa al uso de un entorno de desarrollo integrado (IDE) como Visual Studio o Visual Studio Code.

Puedes obtener la herramienta dotnet instalando el SDK de .NET 6 o mediante la instalación de Visual Studio. Su ubicación predeterminada es %ProgramFiles%\dotnet en Windows o /usr/bin/dotnet en Ubuntu Linux.

Para compilar una aplicación, la herramienta dotnet requiere un archivo de proyecto y uno o más archivos de C#. El siguiente comando crea la estructura básica de un nuevo proyecto de consola:

```
dotnet new Console -n MyFirstProgram
```

Esto crea una subcarpeta llamada MyFirstProgram que contiene un archivo de proyecto llamado MyFirstProgram.csproj y un archivo de C# llamado Program.cs que imprime "Hello world".

Para compilar y ejecutar tu programa, ejecuta el siguiente comando desde la carpeta MyFirstProgram:

```
dotnet run MyFirstProgram
```

O, si solo deseas compilar sin ejecutar:

```
dotnet build MyFirstProgram.csproj
```

El ensamblado de salida se escribirá en una subcarpeta dentro de bin\debug.

Explicamos los ensamblados en detalle en el Capítulo 17.

Sintaxis

La sintaxis de C# está inspirada en la sintaxis de C y C++. En esta sección, describiremos los elementos de sintaxis de C# utilizando el siguiente programa:

```
using System;

int x = 12 * 30;
Console.WriteLine (x);
```

Identificadores y Palabras clave

Los identificadores son los nombres que los programadores eligen para sus clases, métodos, variables, y así sucesivamente. Aquí están los identificadores en nuestro ejemplo de programa, en el orden en que aparecen:

```
System   x   Console   WriteLine
```

Un identificador debe ser una palabra completa, básicamente compuesta por caracteres Unicode que comienzan con una letra o un guión bajo. Los identificadores en C# distinguen entre mayúsculas y minúsculas. Por convención, los parámetros, las variables locales y los campos privados se escriben en notación camel case (por ejemplo, myVariable), y todos los demás identificadores se escriben en notación Pascal case (por ejemplo, MyMethod).

Las palabras clave son nombres que tienen un significado especial para el compilador. Hay dos palabras clave en nuestro ejemplo de programa, using e int.

La mayoría de las palabras clave están reservadas, lo que significa que no se pueden utilizar como identificadores. Aquí está la lista completa de palabras clave reservadas de C#:

```
abstract  do        in          protected   throw
as        double    int         public      true
base      else      interface   readonly    try
bool      enum      internal    record      typeof
break     event     is          ref         uint
byte      explicit  lock        return      ulong
case      extern    long        sbyte       unchecked
catch     false     namespace   sealed      unsafe
char      finally   new         short       ushort
checked   fixed     null        sizeof      using
class     float     object      stackalloc  virtual
const     for       operator    static      void
continue  foreach   out         string      volatile
decimal   goto      override    struct      while
default   if        params      switch
delegate  implicit  private     this
```

Si realmente deseas utilizar un identificador que coincide con una palabra clave reservada, puedes hacerlo calificándolo con el prefijo @; por ejemplo:

```
int using = 123;      // Illegal
int @using = 123;     // Legal
```

El símbolo @ no forma parte del propio identificador. Por lo tanto, @myVariable es lo mismo que myVariable.

Palabras clave contextuales

Algunas palabras clave son contextuales, lo que significa que también puedes usarlas como identificadores sin el símbolo @:

```
add         dynamic  join     on         value
alias       equals   let      or         var
and         from     managed  orderby    with
ascending   get      nameof   partial    when
async       global   nint     remove     where
await       group    not      select     yield
by          init     notnull  set
descending  into     nuint    unmanaged
```

Con las palabras clave contextuales, no puede surgir ambigüedad dentro del contexto en el que se utilizan.

Literales, signos de puntuación y operadores

Los literales son piezas de datos primitivos insertados léxicamente en el programa. Los literales que utilizamos en nuestro ejemplo de programa son 12 y 30.

Los signos de puntuación ayudan a delimitar la estructura del programa. Un ejemplo es el punto y coma, que termina una instrucción. Las instrucciones pueden abarcar varias líneas:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

Un operador transforma y combina expresiones. La mayoría de los operadores en C# se denotan con un símbolo, como el operador de multiplicación, *. Discutiremos los operadores con más detalle más adelante en este capítulo. Estos son los operadores que utilizamos en nuestro ejemplo de programa:

```
=  *  .  ()
```

Un punto denota un miembro de algo (o un punto decimal en literales numéricos). Los paréntesis se utilizan al declarar o llamar a un método; los paréntesis vacíos se utilizan cuando el método no acepta argumentos. (Los paréntesis también tienen otros propósitos que verás más adelante en este capítulo). Un signo igual realiza una asignación. (El doble signo igual, ==, realiza una comparación de igualdad, como verás más adelante).

Comentarios

C# ofrece dos estilos diferentes de documentación en el código fuente: comentarios de una sola línea y comentarios de varias líneas. Un comentario de una sola línea comienza con dos barras diagonales y continúa hasta el final de la línea; por ejemplo:

```
int x = 3;   // Comment about assigning 3 to x
```

Un comentario de varias líneas comienza con /* y termina con /; por ejemplo:

```
int x = 3;   /* This is a comment that
                spans two lines */
```

Los comentarios pueden incluir etiquetas de documentación XML, que explicamos en "Documentación XML" en la página 252.

Conceptos básicos de tipos

Un tipo define el diseño o la estructura de un valor. En este ejemplo, usamos dos literales del tipo int con valores 12 y 30. También declaramos una variable del tipo int con el nombre x:

```
int x = 12 * 30;
Console.WriteLine (x);
```



Debido a que la mayoría de los ejemplos de código en este libro requieren tipos del espacio de nombres System, omitiremos "using System" de ahora en adelante, a menos que estemos ilustrando un concepto relacionado con los espacios de nombres.

Una variable denota una ubicación de almacenamiento que puede contener diferentes valores a lo largo del tiempo. En contraste, una constante siempre representa el mismo valor (más sobre esto más adelante):

```
const int y = 360;
```

Todos los valores en C# son instancias de un tipo. El significado de un valor y el conjunto de valores posibles que puede tener una variable están determinados por su tipo.

Ejemplos de tipos predefinidos

Los tipos predefinidos son tipos que son especialmente compatibles con el compilador. El tipo int es un tipo predefinido para representar el conjunto de enteros que caben en 32 bits de memoria, desde -231 hasta 231 - 1, y es el tipo predeterminado para literales numéricos dentro de este rango. Puede realizar funciones como aritmética con instancias del tipo int, de la siguiente manera:

```
int x = 12 * 30;
```

Otro tipo predefinido en C# es string. El tipo string representa una secuencia de caracteres, como ".NET" o "http://oreilly.com". Puede trabajar con cadenas llamando funciones sobre ellas, de la siguiente manera:

```
  string message = "Hello world";
  string upperMessage = message.ToUpper();
  Console.WriteLine (upperMessage);              // HELLO WORLD

int x = 2022;
message = message + x.ToString();
Console.WriteLine (message);                     // Hello world2022
```

En este ejemplo, llamamos a x.ToString() para obtener una representación de cadena del entero x. Puede llamar a ToString() en una variable de casi cualquier tipo.

El tipo predefinido bool tiene exactamente dos valores posibles: true y false. El tipo bool se usa comúnmente con una declaración if para bifurcar condicionalmente el flujo de ejecución:

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine ("This will not print");

int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("This will print");
```



En C#, los tipos predefinidos (también conocidos como tipos incorporados) se reconocen con una palabra clave de C#. El espacio de nombres System en .NET contiene muchos tipos importantes que no están predefinidos en C# (por ejemplo, DateTime).

Al igual que podemos escribir nuestros propios métodos, también podemos escribir nuestros propios tipos. En el siguiente ejemplo, definimos un tipo personalizado llamado UnitConverter, una clase que sirve como modelo para conversiones de unidades:

```
UnitConverter feetToInchesConverter = new UnitConverter (12);
UnitConverter milesToFeetConverter  = new UnitConverter (5280);

Console.WriteLine (feetToInchesConverter.Convert(30));    // 360
Console.WriteLine (feetToInchesConverter.Convert(100));   // 1200

Console.WriteLine (feetToInchesConverter.Convert(
                   milesToFeetConverter.Convert(1)));     // 63360

public class UnitConverter
{
  int ratio;                              // Field

  public UnitConverter (int unitRatio)    // Constructor
  {
     ratio = unitRatio;
  }

  public int Convert (int unit)           // Method
  {
     return unit * ratio;


  }
}
```



En este ejemplo, nuestra definición de clase aparece en el mismo archivo que nuestras declaraciones de nivel superior. Esto es válido siempre y cuando las declaraciones de nivel superior aparezcan primero, y es aceptable al escribir programas de prueba pequeños. Sin embargo, en programas más grandes, el enfoque estándar es colocar la definición de clase en un archivo separado, como UnitConverter.cs.

Miembros de un tipo

Un tipo contiene miembros de datos y miembros de funciones. El miembro de datos de UnitConverter es el campo llamado "ratio". Los miembros de función de UnitConverter son el método "Convert" y el constructor de UnitConverter.

Simetría entre tipos predefinidos y tipos personalizados

Un aspecto destacado de C# es que los tipos predefinidos y los tipos personalizados tienen pocas diferencias. El tipo predefinido "int" sirve como un modelo para los enteros. Contiene datos, 32 bits, y proporciona miembros de función que utilizan esos datos, como "ToString". De manera similar, nuestro tipo personalizado UnitConverter actúa como un modelo para las conversiones de unidades. Contiene datos, la "ratio", y proporciona miembros de función para utilizar esos datos.

Constructores e instanciación

Los datos se crean mediante la instancia de un tipo. Los tipos predefinidos se pueden instanciar simplemente utilizando un literal como 12 o "Hola mundo". El operador "new" crea instancias de un tipo personalizado. Creamos y declaramos una instancia del tipo UnitConverter con la siguiente declaración:

```
UnitConverter feetToInchesConverter = new UnitConverter (12);
```

Inmediatamente después de que el operador "new" instancia un objeto, se llama al constructor del objeto para realizar la inicialización. Un constructor se define como un método, excepto que el nombre del método y el tipo de retorno se reducen al nombre del tipo que lo contiene:

```
public UnitConverter (int unitRatio) { ratio = unitRatio; }
```

Miembros de instancia versus miembros estáticos

Los miembros de datos y los miembros de funciones que operan en la instancia del tipo se llaman miembros de instancia. El método "Convert" de UnitConverter y el método "ToString" de int son ejemplos de miembros de instancia. Por defecto, los miembros son miembros de instancia.

Los miembros de datos y los miembros de funciones que no operan en la instancia del tipo se pueden marcar como estáticos. Para referirse a un miembro estático desde fuera de su tipo, se especifica el nombre del tipo en lugar de una instancia. Un ejemplo es el método "WriteLine" de la clase Console. Debido a que es estático, llamamos a Console.WriteLine() y no a new Console().WriteLine().

(La clase Console se declara como una clase estática, lo que significa que todos sus miembros son estáticos y nunca se pueden crear instancias de Console.)

En el siguiente código, el campo de instancia "Name" se refiere a una instancia particular de un Panda, mientras que "Population" se refiere al conjunto de todas las instancias de Panda. Creamos dos instancias de Panda, imprimimos sus nombres y luego imprimimos la población total:

```
Panda p1 = new Panda ("Pan Dee");
Panda p2 = new Panda ("Pan Dah");

Console.WriteLine (p1.Name);      // Pan Dee
Console.WriteLine (p2.Name);      // Pan Dah

Console.WriteLine (Panda.Population);   // 2

public class Panda
{
  public string Name;             // Instance field
  public static int Population;   // Static field

  public Panda (string n)         // Constructor
  {
    Name = n;                     // Assign the instance field
    Population = Population + 1;  // Increment the static Population field
  }
}
```

Intentar evaluar p1.Population o Panda.Name generará un error en tiempo de compilación.

La palabra clave public

La palabra clave public expone miembros a otras clases. En este ejemplo, si el campo Name en Panda no estuviera marcado como public, sería privado y no se podría acceder desde fuera de la clase. Marcar un miembro como public es la forma en que un tipo se comunica diciendo: "Aquí está lo que quiero que otros tipos vean; todo lo demás son detalles de implementación privados". En términos de programación orientada a objetos, decimos que los miembros públicos encapsulan los miembros privados de la clase.

Definición de espacios de nombres

Especialmente en programas más grandes, tiene sentido organizar tipos en espacios de nombres. Así es como se define la clase Panda dentro de un espacio de nombres llamado Animals:

```
using System;
using Animals;

Panda p = new Panda ("Pan Dee");
Console.WriteLine (p.Name);

namespace Animals
{
  public class Panda


  {
     ...
  }
}
```

En este ejemplo, también importamos el espacio de nombres Animals para que nuestras declaraciones de nivel superior puedan acceder a sus tipos sin calificación. Sin esa importación, tendríamos que hacer esto:

```
Animals.Panda p = new Animals.Panda ("Pan Dee");
```

En este capítulo, cubriremos los espacios de nombres en detalle al final (ver "Espacios de nombres" en la página 87).

Definición de un método Main

Hasta ahora, todos nuestros ejemplos han utilizado declaraciones de nivel superior (una característica introducida en C# 9).

Sin declaraciones de nivel superior, una aplicación de consola o de Windows simple se vería así:

```
using System;

class Program
{
  static void Main()   // Program entry point
  {
    int x = 12 * 30;
    Console.WriteLine (x);
  }
}
```

En ausencia de declaraciones de nivel superior, C# busca un método estático llamado Main, que se convierte en el punto de entrada. El método Main se puede definir dentro de cualquier clase (y solo puede existir un método Main). Si el método Main necesita acceder a miembros privados de una clase en particular, definir un método Main dentro de esa clase puede ser más sencillo que usar declaraciones de nivel superior.

El método Main puede opcionalmente devolver un entero (en lugar de void) para devolver un valor al entorno de ejecución (donde un valor distinto de cero normalmente indica un error). El método Main también puede opcionalmente aceptar un array de cadenas como parámetro (que se llenará con cualquier argumento pasado al ejecutable).

Por ejemplo:

```
static int Main (string[] args) {...}
```



Un array (como string[]) representa un número fijo de elementos de un tipo particular. Los arrays se especifican colocando corchetes cuadrados después del tipo de elemento. Los describimos en "Arrays" en la página 54.

(El método Main también puede declararse como async y devolver una Task o Task<int> para admitir la programación asíncrona, que cubrimos en el Capítulo 14.)

Top-Level Statements

Top-level statements (introduced in C# 9) let you avoid the baggage of a static `Main` method and a containing class. A file with top-level statements comprises three parts, in this order:

1. (Optionally) `using` directives
2. A series of statements, optionally mixed with method declarations
3. (Optionally) Type and namespace declarations

For example:

```
using System;                           // Part 1

Console.WriteLine ("Hello, world");     // Part 2
void SomeMethod1() { ... }              // Part 2
Console.WriteLine ("Hello again!");     // Part 2
void SomeMethod2() { ... }              // Part 2

class SomeClass { ... }                 // Part 3
namespace SomeNamespace { ... }         // Part 3
```

Because the CLR doesn't explicitly support top-level statements, the compiler translates your code into something like this:

```
using System;                           // Part 1

static class Program$   // Special compiler-generated name
{
   static void Main$ (string[] args)   // Compiler-generated name
   {
    Console.WriteLine ("Hello, world");     // Part 2
    void SomeMethod1() { ... }              // Part 2
    Console.WriteLine ("Hello again!");     // Part 2
    void SomeMethod2() { ... }              // Part 2
  }
 }

class SomeClass { ... }                 // Part 3
namespace SomeNamespace { ... }         // Part 3
```

Notice that everything in Part 2 is wrapped inside the main method. This means that `SomeMethod1` and `SomeMethod2` act as *local methods*. We discuss the full implications in "Local methods" on page 98, the most important being that local methods (unless declared as `static`) can access variables declared within the containing method:

```
int x = 3;
LocalMethod();

void LocalMethod() { Console.WriteLine (x); }   // We can access x
```

Another consequence is that top-level methods cannot be accessed from other classes or types.

Top-level statements can optionally return an integer value to the caller and access a "magic" variable of type `string[]` called `args`, corresponding to command-line arguments passed by the caller.

As a program can have only one entry point, there can be at most one file with top-level statements in a C# project.

Tipos y Conversiones

C# puede convertir entre instancias de tipos compatibles. Una conversión siempre crea un nuevo valor a partir de uno existente. Las conversiones pueden ser implícitas o explícitas: las conversiones implícitas ocurren automáticamente y las conversiones explícitas requieren una operación de tipo conocida como "casting".

En el siguiente ejemplo, convertimos implícitamente un entero (int) a un entero largo (long) (que tiene el doble de capacidad de bits que un entero) y convertimos explícitamente un entero (int) a un entero corto (short) (que tiene la mitad de capacidad de bits de un entero):

```
int x = 12345;       // int is a 32-bit integer
long y = x;          // Implicit conversion to 64-bit integer
short z = (short)x;  // Explicit conversion to 16-bit integer
```

Las conversiones implícitas están permitidas cuando se cumplen ambas condiciones siguientes:

El compilador puede garantizar que siempre tendrán éxito.

No se pierde información durante la conversión.

Por el contrario, las conversiones explícitas son necesarias cuando se cumple una de las siguientes condiciones:

El compilador no puede garantizar que siempre tendrán éxito.

Puede perderse información durante la conversión.

(En caso de que el compilador determine que una conversión siempre fallará, ambos tipos de conversión están prohibidos. Las conversiones que involucran tipos genéricos también pueden fallar en ciertas condiciones; consulta "Parámetros de tipo y conversiones" en la página 152).



Las conversiones numéricas que acabamos de ver están integradas en el lenguaje. C# también admite conversiones de referencia y conversiones de boxing (consultar Capítulo 3), así como conversiones personalizadas (consultar "Sobrecarga de operadores" en la página 239). El compilador no aplica las reglas mencionadas anteriormente con las conversiones personalizadas, por lo que es posible que los tipos mal diseñados se comporten de manera diferente.

Tipos de valor versus tipos de referencia

Todos los tipos en C# se dividen en las siguientes categorías:

• Tipos de valor

• Tipos de referencia

• Parámetros de tipo genérico

• Tipos de puntero



En esta sección, describimos los tipos de valor y los tipos de referencia. Cubrimos los parámetros de tipo genérico en "Genéricos" en la página 145 y los tipos de puntero en "Código inseguro y punteros" en la página 243.

Los tipos de valor comprenden la mayoría de los tipos integrados (específicamente, todos los tipos numéricos, el tipo char y el tipo bool) así como los tipos de estructura (struct) y enumeración (enum) personalizados.

Los tipos de referencia comprenden todos los tipos de clase (class), matriz (array), delegado (delegate) e interfaz (interface). Esto incluye el tipo predefinido string.

La diferencia fundamental entre los tipos de valor y los tipos de referencia es cómo se manejan en la memoria.

Tipos de valor:

El contenido de una variable o constante de tipo de valor es simplemente un valor. Por ejemplo, el contenido del tipo de valor integrado int es de 32 bits de datos.

Puedes definir un tipo de valor personalizado utilizando la palabra clave struct (estructura) (ver Figura 2-1):

```
public struct Point { public int X; public int Y; }
```

De manera más concisa:

```
public struct Point { public int X, Y; }
```

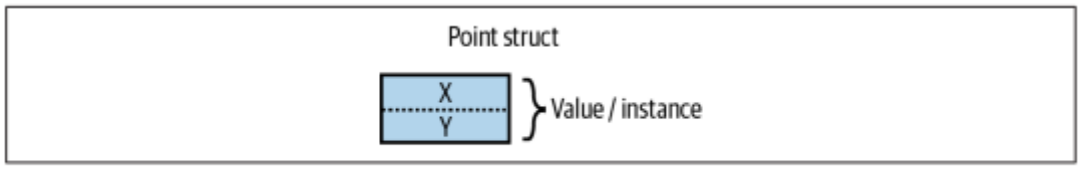


Figure 2-1. A value-type instance in memory

La asignación de una instancia de un tipo de valor siempre copia la instancia; por ejemplo:

```
Point p1 = new Point();
p1.X = 7;

Point p2 = p1;             // Assignment causes copy

Console.WriteLine (p1.X);  // 7
Console.WriteLine (p2.X);  // 7
```

```
p1.X = 9;                  // Change p1.X

Console.WriteLine (p1.X);  // 9
Console.WriteLine (p2.X);  // 7
```

Figure 2-2 shows that p1 and p2 have independent storage.

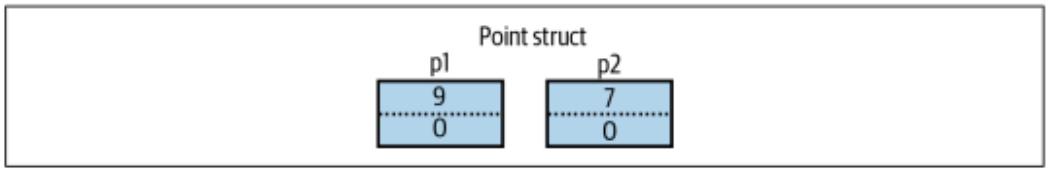


Figure 2-2. Assignment copies a value-type instance

Tipos de referencia

Un tipo de referencia es más complejo que un tipo de valor, ya que tiene dos partes: un objeto y la referencia a ese objeto. El contenido de una variable o constante de tipo de referencia es una referencia a un objeto que contiene el valor. Aquí está el tipo Point de nuestro ejemplo anterior reescrito como una clase en lugar de una estructura (mostrado en la Figura 2-3):

```
public class Point { public int X, Y; }
```

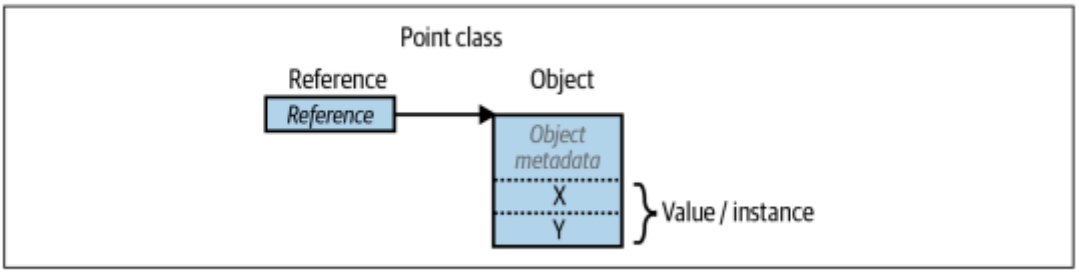


Figure 2-3. A reference-type instance in memory

La asignación de una variable de tipo de referencia copia la referencia, no la instancia del objeto. Esto permite que múltiples variables se refieran al mismo objeto, algo que no es posible de forma ordinaria con los tipos de valor. Si repetimos el ejemplo anterior, pero ahora con Point como una clase, una operación en p1 afectará a p2:

```
Point p1 = new Point();
p1.X = 7;

Point p2 = p1;             // Copies p1 reference

Console.WriteLine (p1.X);  // 7
Console.WriteLine (p2.X);  // 7

p1.X = 9;                  // Change p1.X

Console.WriteLine (p1.X);  // 9
Console.WriteLine (p2.X);  // 9
```

Figure 2-4 shows that p1 and p2 are two references that point to the same object.

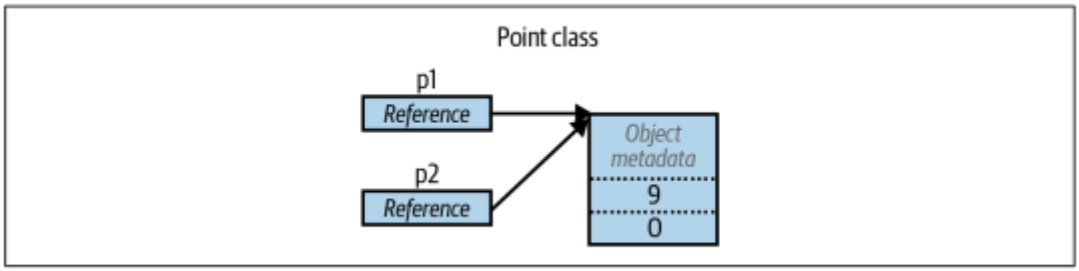


Figure 2-4. Assignment copies a reference

Null

Una referencia puede ser asignada al valor literal null, indicando que la referencia no apunta a ningún objeto:

```
Point p = null;
Console.WriteLine (p == null);   // True

// The following line generates a runtime error
// (a NullReferenceException is thrown):
Console.WriteLine (p.X);

class Point {...}
```



En "Nullable Reference Types" en la página 200, describimos una característica de C# que ayuda a reducir los errores accidentales de NullReference Exception.

En contraste, un tipo de valor no puede tener de forma ordinaria un valor nulo:

```
Point p = null;  // Compile-time error
int x = null;    // Compile-time error

struct Point {...}
```



C# también tiene un constructo llamado tipos de valor nulables para representar nulos de tipo de valor. Para obtener más información, consulta "Tipos de valor nulables" en la página 194.

El almacenamiento adicional

Las instancias de tipos de valor ocupan exactamente la memoria necesaria para almacenar sus campos. En este ejemplo, Point ocupa 8 bytes de memoria:

```
struct Point
{
  int x;  // 4 bytes
  int y;  // 4 bytes
}
```



Técnicamente, el CLR coloca los campos dentro del tipo en una dirección que es un múltiplo del tamaño de los campos (hasta un máximo de 8 bytes). Por lo tanto, lo siguiente consume en realidad 16 bytes de memoria (con los 7 bytes siguientes al primer campo "desperdiciados"):

```
struct A { byte b; long l; }
```

Puedes anular este comportamiento aplicando el atributo StructLayout (consultar "Mapeo de una estructura a memoria no administrada" en la página 973).

Los tipos de referencia requieren asignaciones separadas de memoria para la referencia y el objeto. El objeto consume tantos bytes como sus campos, más una sobrecarga administrativa adicional. La sobrecarga precisa es intrínsecamente privada a la implementación del tiempo de ejecución de .NET, pero como mínimo, la sobrecarga es de 8 bytes, utilizados para almacenar una clave del tipo del objeto, así como información temporal como su estado de bloqueo para la multiprocesamiento y una bandera para indicar si ha sido fijado para evitar su movimiento por parte del recolector de basura. Cada referencia a un objeto requiere un extra de 4 o 8 bytes, dependiendo de si el tiempo de ejecución de .NET se está ejecutando en una plataforma de 32 o 64 bits.

Taxonomía de tipos predefinidos

Los tipos predefinidos en C# son los siguientes:

Tipos de valor

• Numéricos

— Entero con signo (sbyte, short, int, long)

— Entero sin signo (byte, ushort, uint, ulong)

— Número real (float, double, decimal)

• Lógico (bool)

• Carácter (char)

Tipos de referencia

• Cadena de texto (string)

• Objeto genérico (object)

Los tipos predefinidos en C# hacen alias de los tipos de .NET en el espacio de nombres System. Solo hay una diferencia sintáctica entre estas dos declaraciones:

```
int i = 5;
System.Int32 i = 5;
```

El conjunto de tipos de valor predefinidos, excluyendo decimal, se conocen como tipos primitivos en el CLR. Los tipos primitivos se llaman así porque se admiten directamente mediante instrucciones en el código compilado, y esto suele traducirse en un soporte directo en el procesador subyacente; por ejemplo:

```
                    // Underlying hexadecimal representation
int i = 7;          // 0x7
bool b = true;      // 0x1
char c = 'A';       // 0x41
float f = 0.5f;     // uses IEEE floating-point encoding
```

Los tipos System.IntPtr y System.UIntPtr también son primitivos (consultar Capítulo 24).

Tipos numéricos

C# tiene los tipos numéricos predefinidos que se muestran en la Tabla 2-1.

Tabla 2-1. Tipos numéricos predefinidos en C#

Table 2-1. Predefined numeric types in C#

C# type	System type	Suffix	Size	Range
Integral—signed				
sbyte	SByte		8 bits	-2^7 to 2^7-1
short	Int16		16 bits	-2^{15} to $2^{15}-1$
int	Int32		32 bits	-2^{31} to $2^{31}-1$
long	Int64	L	64 bits	-2^{63} to $2^{63}-1$
nint	IntPtr		32/64 bits	
Integral—unsigned				
byte	Byte		8 bits	0 to 2^8-1
ushort	UInt16		16 bits	0 to $2^{16}-1$
uint	UInt32	U	32 bits	0 to $2^{32}-1$
ulong	UInt64	UL	64 bits	0 to $2^{64}-1$
unint	UIntPtr		32/64 bits	
Real				
float	Single	F	32 bits	$\pm(\sim10^{-45}$ to $10^{38})$
double	Double	D	64 bits	$\pm(\sim10^{-324}$ to $10^{308})$
decimal	Decimal	M	128 bits	$\pm(\sim10^{-28}$ to $10^{28})$

De los tipos enteros, int y long son ciudadanos de primera clase y son preferidos tanto por C# como por el tiempo de ejecución. Los otros tipos enteros se utilizan típicamente para interoperabilidad o cuando la eficiencia de espacio es primordial. Los tipos de entero nativo nint y nuint (introducidos en C# 9) son más útiles para ayudar con la aritmética de punteros, por lo que los describiremos en un capítulo posterior (consultar "Enteros de Tamaño Nativo" en la página 246).

De los tipos de números reales, float y double se llaman tipos de punto flotante y se utilizan típicamente para cálculos científicos y gráficos. El tipo decimal se utiliza típicamente para cálculos financieros, donde se requiere una aritmética precisa en base 10 y una alta precisión.



A partir de .NET 5, existe un tipo de punto flotante de 16 bits llamado Half. Esto está destinado principalmente a la interoperabilidad con procesadores de tarjetas gráficas y no cuenta con soporte nativo en la mayoría de las CPUs. Half no es un tipo primitivo del CLR y no cuenta con un soporte especial en el lenguaje C#.

Literales numéricos

Los literales de tipos enteros pueden utilizar notación decimal o hexadecimal; la notación hexadecimal se indica con el prefijo 0x. Por ejemplo:

```
int x = 127;
long y = 0x7F;
```

Puedes insertar un guión bajo en cualquier lugar dentro de un literal numérico para hacerlo más legible:

```
int million = 1_000_000;
```

Puedes especificar números en binario con el prefijo 0b:

```
var b = 0b1010_1011_1100_1101_1110_1111;
```

Los literales reales pueden utilizar notación decimal y/o exponencial:

```
double d = 1.5;
double million = 1E06;
```

Inferencia de tipo de literales numéricos

Por defecto, el compilador infiere que un literal numérico es de tipo double o de un tipo entero:

- Si el literal contiene un punto decimal o el símbolo exponencial (E), es de tipo double.
- De lo contrario, el tipo del literal es el primer tipo de esta lista que puede contener el valor del literal: int, uint, long y ulong.

Por ejemplo:

```
Console.WriteLine (        1.0.GetType());  // Double  (double)
Console.WriteLine (       1E06.GetType());  // Double  (double)
Console.WriteLine (          1.GetType());  // Int32   (int)


Console.WriteLine ( 0xF0000000.GetType());  // UInt32  (uint)
Console.WriteLine (0x100000000.GetType());  // Int64   (long)
```

Sufijos numéricos

Los sufijos numéricos definen explícitamente el tipo de un literal. Los sufijos pueden ser en minúsculas o mayúsculas y son los siguientes:

Category	C# type	Example
F	float	float f = 1.0F;
D	double	double d = 1D;
M	decimal	decimal d = 1.0M;
U	uint	uint i = 1U;
L	long	long i = 1L;
UL	ulong	ulong i = 1UL;

Los sufijos U y L rara vez son necesarios porque los tipos uint, long y ulong casi siempre se pueden inferir o convertir implícitamente desde int:

```
long i = 5;     // Implicit lossless conversion from int literal to long
```

El sufijo D es técnicamente redundante, ya que todos los literales con un punto decimal se infieren como double. Y siempre se puede agregar un punto decimal a un literal numérico:

```
double x = 4.0;
```

Los sufijos F y M son los más útiles y se deben aplicar siempre al especificar literales de float o decimal. Sin el sufijo F, la siguiente línea no se compilaría, ya que 4.5 se inferiría como tipo double, que no tiene una conversión implícita a float:

```
float f = 4.5F;
```

El mismo principio es válido para un literal decimal:

```
decimal d = -1.23M;     // Will not compile without the M suffix.
```

Describimos en detalle la semántica de las conversiones numéricas en la siguiente sección.

Conversiones numéricas

Conversión entre tipos enteros

Las conversiones de tipo entero son implícitas cuando el tipo de destino puede representar todos los posibles valores del tipo de origen. De lo contrario, se requiere una conversión explícita; por ejemplo:

```
int x = 12345;       // int is a 32-bit integer
long y = x;          // Implicit conversion to 64-bit integral type
short z = (short)x;  // Explicit conversion to 16-bit integral type
```

La conversión entre tipos de punto flotante

Un float se puede convertir implícitamente a double dado que un double puede representar todos los posibles valores de un float. La conversión inversa debe ser explícita.

Conversión entre tipos de punto flotante y enteros

Todos los tipos enteros se pueden convertir implícitamente a todos los tipos de punto flotante:

```
int i = 1;
float f = i;
```

La conversión inversa debe ser explícita:

```
int i2 = (int)f;
```



Cuando realizas una conversión de un número de punto flotante a un tipo entero, cualquier parte fraccionaria se trunca; no se realiza ningún redondeo. La clase estática System.Convert proporciona métodos que realizan un redondeo al convertir entre diferentes tipos numéricos (ver Capítulo 6).

Al convertir implícitamente un tipo entero grande a un tipo de punto flotante se preserva la magnitud, pero ocasionalmente se puede perder precisión. Esto se debe a que los tipos de punto flotante siempre tienen una mayor magnitud que los tipos enteros, pero pueden tener menos precisión. Si reescribimos nuestro ejemplo con un número más grande, se puede apreciar esto:

```
int i1 = 100000001;
float f = i1;          // Magnitude preserved, precision lost
int i2 = (int)f;       // 100000000
```

Conversiones decimales

Todos los tipos enteros se pueden convertir implícitamente al tipo decimal, dado que un decimal puede representar cualquier valor posible de los tipos enteros en C#. Todas las demás conversiones numéricas hacia y desde un tipo decimal deben ser explícitas, ya que introducen la posibilidad de que un valor esté fuera de rango o se pierda precisión.

Operadores aritméticos

Los operadores aritméticos (+, -, *, /, %) están definidos para todos los tipos numéricos excepto los tipos enteros de 8 y 16 bits.

```
+    Addition
-    Subtraction
*    Multiplication
/    Division
%    Remainder after division
```

Operadores de incremento y decremento

Los operadores de incremento (++), y decremento (--), incrementan y decrementan los tipos numéricos en 1. El operador puede seguir o preceder a la variable, dependiendo de si quieres obtener su valor antes o después del incremento/decremento. Por ejemplo:

```
int x = 0, y = 0;
Console.WriteLine (x++);   // Outputs 0; x is now 1
Console.WriteLine (++y);   // Outputs 1; y is now 1
```

Operaciones especializadas en tipos enteros

Los tipos enteros son int, uint, long, ulong, short, ushort, byte y sbyte.

División

Las operaciones de división en tipos enteros siempre eliminan el resto (redondean hacia cero). Dividir por una variable cuyo valor es cero genera un error en tiempo de ejecución (una DivideByZeroException):

```
int a = 2 / 3;      // 0

int b = 0;
int c = 5 / b;      // throws DivideByZeroException
```

Dividir por el literal o constante 0 genera un error en tiempo de compilación.

Desbordamiento

En tiempo de ejecución, las operaciones aritméticas en tipos enteros pueden desbordarse. De forma predeterminada, esto ocurre de manera silenciosa, no se lanza ninguna excepción y el resultado muestra un comportamiento de "envoltura", como si el cálculo se realizara en un tipo entero más grande y los bits significativos adicionales se descartaran. Por ejemplo, decrementar el valor mínimo posible de int resulta en el valor máximo posible de int:

```
int a = int.MinValue;
a--;
Console.WriteLine (a == int.MaxValue); // True
```

Operadores de verificación de desbordamiento

El operador checked instruye al tiempo de ejecución a generar una OverflowException en lugar de desbordarse silenciosamente cuando una expresión o instrucción de tipo entero excede los límites aritméticos de ese tipo. El operador checked afecta a las expresiones con los operadores ++, --, +, - (binarios y unarios), *, /, y a los operadores de conversión explícita entre tipos enteros. La verificación de desbordamiento incurre en un pequeño costo de rendimiento.



El operador checked no tiene efecto en los tipos double y float (que se desbordan a valores especiales "infinitos", como verás pronto) y no tiene efecto en el tipo decimal (que siempre se verifica).

Puedes usar checked alrededor de una expresión o un bloque de instrucciones:

```
int a = 1000000;
int b = 1000000;

int c = checked (a * b);      // Checks just the expression.

checked                       // Checks all expressions
{                             // in statement block.
   ...
   c = a * b;
   ...
}
```

Puedes hacer que la verificación de desbordamiento aritmético sea la configuración predeterminada para todas las expresiones en un programa seleccionando la opción "checked" a nivel de proyecto (en Visual Studio, ve a Configuración avanzada de compilación). Si luego necesitas desactivar la verificación de desbordamiento solo para expresiones o declaraciones específicas, puedes hacerlo con el operador unchecked. Por ejemplo, el siguiente código no lanzará excepciones, incluso si la opción "checked" del proyecto está seleccionada:

```
int x = int.MaxValue;
int y = unchecked (x + 1);
unchecked { int z = x + 1; }
```

Sin importar la configuración "checked" del proyecto, las expresiones evaluadas en tiempo de compilación siempre se verifican para desbordamientos, a menos que apliques el operador unchecked:

```
int x = int.MaxValue + 1;               // Compile-time error
int y = unchecked (int.MaxValue + 1);   // No errors
```

Operadores bit a bit

Los operadores bit a bit que C# admite son los siguientes:

Operator	Meaning	Sample expression	Result
~	Complement	~0xfU	0xfffffff0U
&	And	0xf0 & 0x33	0x30
|	Or	0xf0 | 0x33	0xf3
^	Exclusive Or	0xff00 ^ 0x0ff0	0xf0f0
<<	Shift left	0x20 << 2	0x80
>>	Shift right	0x20 >> 1	0x10



A partir de .NET 6, se exponen operaciones adicionales de bits a través de una nueva clase llamada BitOperations en el espacio de nombres System.Numerics (ver "BitOperations" en la página 320).

Tipos enteros de 8 y 16 bits

Los tipos enteros de 8 y 16 bits son byte, sbyte, short y ushort. Estos tipos no tienen sus propios operadores aritméticos, por lo que C# los convierte implícitamente a tipos más grandes según sea necesario. Esto puede causar un error en tiempo de compilación al intentar asignar el resultado de vuelta a un tipo entero pequeño:

```
short x = 1, y = 1;
short z = x + y;          // Compile-time error
```

En este caso, x e y se convierten implícitamente a int para que se pueda realizar la adición. Esto significa que el resultado también es un int, que no se puede convertir implícitamente de vuelta a un short (porque podría provocar pérdida de datos). Para que esto compile, debes agregar una conversión explícita:

```
short z = (short) (x + y);   // OK
```

Valores Especiales de Float y Double

A diferencia de los tipos enteros, los tipos de punto flotante tienen valores que ciertas operaciones tratan de manera especial. Estos valores especiales son NaN (No es un número), +∞, −∞ y −0. Las clases float y double tienen constantes para NaN, +∞ y −∞, así como otros valores (MaxValue, MinValue y Epsilon); por ejemplo:

```
Console.WriteLine (double.NegativeInfinity);   // -Infinity
```

Las constantes que representan valores especiales para double y float son las siguientes:

Special value	Double constant	Float constant
NaN	double.NaN	float.NaN
+∞	double.PositiveInfinity	float.PositiveInfinity
−∞	double.NegativeInfinity	float.NegativeInfinity
−0	-0.0	-0.0f

Dividir un número distinto de cero por cero da como resultado un valor infinito:

```
Console.WriteLine ( 1.0 /  0.0);                //  Infinity
Console.WriteLine (-1.0 /  0.0);                // -Infinity
Console.WriteLine ( 1.0 / -0.0);                // -Infinity
Console.WriteLine (-1.0 / -0.0);                //  Infinity
```

Dividir cero por cero, o restar infinito de infinito, da como resultado un NaN:

```
Console.WriteLine ( 0.0 /  0.0);                 //  NaN
Console.WriteLine ((1.0 /  0.0) - (1.0 / 0.0));  //  NaN
```

Cuando se usa ==, un valor de NaN nunca es igual a otro valor, ni siquiera a otro NaN

valor:

```
Console.WriteLine (0.0 / 0.0 == double.NaN);    // False
```

Para probar si un valor es NaN, debe usar float.IsNaN o double.IsNaN

método:

```
Console.WriteLine (double.IsNaN (0.0 / 0.0));   // True
```

Sin embargo, cuando se usa object.Equals, dos valores de NaN son iguales:

```
Console.WriteLine (object.Equals (0.0 / 0.0, double.NaN));   // True
```



Los NaN a veces son útiles para representar valores especiales. En Windows Presentation Foundation (WPF), double.NaN representa una medida cuyo valor es "Automático". Otra forma de representar dicho valor es con un tipo nullable (Capítulo 4); otra opción es con una estructura personalizada que envuelve un tipo numérico y agrega un campo adicional (Capítulo 3).

float y double siguen la especificación de los tipos de formato IEEE 754, soportados nativamente por casi todos los procesadores. Puedes encontrar información detallada sobre el comportamiento de estos tipos en http://www.ieee.org.

double frente a decimal

double es útil para cálculos científicos (como el cálculo de coordenadas espaciales). decimal es útil para cálculos financieros y valores que son "hechos por humanos" en lugar de ser el resultado de mediciones del mundo real. Aquí tienes un resumen de las diferencias:

Category	double	decimal
Internal representation	Base 2	Base 10
Decimal precision	15–16 significant figures	28–29 significant figures
Range	$\pm(\sim10^{-324}$ to $\sim10^{308})$	$\pm(\sim10^{-28}$ to $\sim10^{28})$
Special values	+0, −0, +∞, −∞, and NaN	None
Speed	Native to processor	Non-native to processor (about 10 times slower than double)

Errores de redondeo de números reales

float y double representan internamente números en base 2. Por esta razón, solo se pueden representar de forma precisa números expresables en base 2. En la práctica, esto significa que la mayoría de los literales con componente fraccionaria (que están en base 10) no se representarán de forma precisa; por ejemplo:

```
float x = 0.1f;  // Not quite 0.1
Console.WriteLine (x + x + x + x + x + x + x + x + x + x);    // 1.0000001
```

Esta es la razón por la cual float y double no son adecuados para cálculos financieros. En cambio, decimal trabaja en base 10 y puede representar de forma precisa números expresables en base 10 (así como sus factores, base 2 y base 5). Debido a que los literales reales están en base 10, decimal puede representar de forma precisa números como 0.1. Sin embargo, ni double ni decimal pueden representar de manera precisa un número fraccionario cuya representación en base 10 es recurrente.

```
decimal m = 1M / 6M;              // 0.16666666666666666666666666667M
double  d = 1.0 / 6.0;            // 0.16666666666666666
```

Esto lleva a la acumulación de errores de redondeo:

```
decimal notQuiteWholeM = m+m+m+m+m+m;  // 1.00000000000000000000000002M
double  notQuiteWholeD = d+d+d+d+d+d;  // 0.99999999999999989
```

lo cual rompe las operaciones de igualdad y comparación:

```
Console.WriteLine (notQuiteWholeM == 1M);   // False
Console.WriteLine (notQuiteWholeD < 1.0);   // True
```

Tipo booleano y operadores

El tipo bool de C# (alias del tipo System.Boolean) es un valor lógico que puede ser asignado con el literal true o false.

Aunque un valor booleano solo requiere 1 bit de almacenamiento, el tiempo de ejecución utiliza 1 byte de memoria debido a que es el tamaño mínimo con el que el tiempo de ejecución y el procesador pueden trabajar eficientemente. Para evitar la ineficiencia de espacio en caso de arreglos, .NET provee una clase llamada BitArray en el espacio de nombres System.Collections que está diseñada para utilizar solo 1 bit por valor booleano.

Conversiones del tipo bool

No se pueden realizar conversiones explícitas del tipo bool a tipos numéricos, ni viceversa.

Operadores de igualdad y comparación

Los operadores == y != se utilizan para probar la igualdad y desigualdad de cualquier tipo, pero siempre retornan un valor bool. Los tipos de valor suelen tener una noción muy simple de igualdad:

```
int x = 1;
int y = 2;
int z = 1;
Console.WriteLine (x == y);         // False
Console.WriteLine (x == z);         // True
```

Para los tipos de referencia, la igualdad por defecto se basa en la referencia, en lugar del valor real del objeto subyacente (más información sobre esto en el Capítulo 6):

```
Dude d1 = new Dude ("John");
Dude d2 = new Dude ("John");
Console.WriteLine (d1 == d2);       // False
Dude d3 = d1;


   Console.WriteLine (d1 == d3);       // True

   public class Dude
   {
     public string Name;
     public Dude (string n) { Name = n; }
   }
```

Los operadores de igualdad y comparación, ==, !=, <, >, >= y <=, funcionan para todos los tipos numéricos, pero debes usarlos con precaución con los números reales (como vimos en "Errores de redondeo de números reales" en la página 48). Los operadores de comparación también funcionan en los miembros de tipos enumerados al comparar sus valores subyacentes de tipo integral. Esto se describe en "Enumeraciones" en la página 140.

Explicamos los operadores de igualdad y comparación en mayor detalle en "Sobrecarga de operadores" en la página 239, "Comparación de igualdad" en la página 324 y "Comparación de orden" en la página 335.

Operadores condicionales

Los operadores && y || se utilizan para probar condiciones de "y" y "o", respectivamente. A menudo se usan junto con el operador !, que representa la negación. En el siguiente ejemplo, el método UseUmbrella devuelve true si está lloviendo o soleado (para protegernos de la lluvia o del sol), siempre y cuando no esté ventoso (los paraguas son inútiles en el viento):

```
static bool UseUmbrella (bool rainy, bool sunny, bool windy)
{
  return !windy && (rainy || sunny);
}
```

Los operadores && y || realizan una evaluación de cortocircuito cuando es posible. En el ejemplo anterior, si está ventoso, la expresión (rainy || sunny) ni siquiera se evalúa.

El cortocircuito es esencial para permitir que expresiones como la siguiente se ejecuten sin lanzar una excepción NullReferenceException:

```
if (sb != null && sb.Length > 0) ...
```

Los operadores & y | también prueban condiciones de "y" y "o", respectivamente:

```
return !windy & (rainy | sunny);
```

La diferencia es que no realizan una evaluación de cortocircuito. Por esta razón, rara vez se utilizan en lugar de los operadores condicionales.



A diferencia de C y C++, los operadores & y | realizan comparaciones booleanas (sin cortocircuito) cuando se aplican a expresiones bool. Los operadores & y | realizan operaciones a nivel de bits solo cuando se aplican a números.

Operador condicional (operador ternario)

El operador condicional (comúnmente conocido como operador ternario porque es el único operador que toma tres operandos) tiene la forma q ? a : b; por lo tanto, si la condición q es verdadera, se evalúa a, de lo contrario se evalúa b:

```
static int Max (int a, int b)
{
  return (a > b) ? a : b;
}
```

El operador condicional es especialmente útil en expresiones LINQ (Capítulo 8).

Cadenas de texto y caracteres

El tipo char de C# (alias del tipo System.Char) representa un carácter Unicode y ocupa 2 bytes (UTF-16). Un literal de tipo char se especifica entre comillas simples:

```
char c = 'A';       // Simple character
```

Las secuencias de escape se utilizan para representar caracteres que no se pueden expresar o interpretar literalmente. Una secuencia de escape consiste en una barra invertida seguida de un carácter con un significado especial; por ejemplo:

```
char newLine = '\n';
char backSlash = '\\';
```

Table 2-2 shows the escape sequence characters.

Table 2-2. Escape sequence characters

Char	Meaning	Value
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

La secuencia de escape \u (o \x) te permite especificar cualquier carácter Unicode mediante su código hexadecimal de cuatro dígitos:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

Char conversiones

Las conversiones implícitas de char a un tipo numérico funcionan para los tipos numéricos que pueden alojar un unsigned short. Para otros tipos numéricos, se requiere una conversión explícita.

Tipo String

El tipo string de C# (alias del tipo System.String, cubierto en profundidad en el Capítulo 6) representa una secuencia inmutable (no modificable) de caracteres Unicode. Un literal de string se especifica entre comillas dobles:

```
string a = "Heat";
```



string es un tipo de referencia en lugar de un tipo de valor. Sin embargo, sus operadores de igualdad siguen la semántica de los tipos de valor:

```
string a = "test";
string b = "test";
Console.Write (a == b);  // True
```

Las secuencias de escape que son válidas para los literales de caracteres también funcionan dentro de las cadenas de texto:

```
string a = "Here's a tab:\t";
```

El costo de esto es que cada vez que necesites un carácter de barra invertida literal, debes escribirlo dos veces:

```
string a1 = "\\\\server\\fileshare\\helloworld.cs";
```

Para evitar este problema, C# permite literales de cadena de texto con formato. Un literal de cadena de texto con formato se precede con el símbolo @ y no admite secuencias de escape. La siguiente cadena de texto con formato es idéntica a la anterior:

```
string a2 = @"\\server\fileshare\helloworld.cs";
```

Un literal de cadena de texto con formato también puede abarcar varias líneas:

```
string escaped  = "First Line\r\nSecond Line";
string verbatim = @"First Line
Second Line";

// True if your text editor uses CR-LF line separators:
Console.WriteLine (escaped == verbatim);
```

Puedes incluir el carácter de comilla doble en un literal de cadena de texto con formato escribiéndolo dos veces:

```
string xml = @"<customer id=""123""></customer>";
```

String concatenacion

La concatenación de cadenas se realiza con el operador +:

```
string s = "a" + "b";
```

Uno de los operandos puede ser un valor que no es una cadena, en cuyo caso se llama al método ToString en ese valor:

```
string s = "a" + 5;  // a5
```

Usar repetidamente el operador + para construir una cadena es ineficiente: una mejor solución es utilizar el tipo System.Text.StringBuilder (descrito en el Capítulo 6).

Interpolación de cadenas

Una cadena precedida del carácter $ se llama una cadena interpolada. Las cadenas interpoladas pueden incluir expresiones encerradas entre llaves:

```
int x = 4;
Console.Write ($"A square has {x} sides");  // Prints: A square has 4 sides
```

Cualquier expresión válida de cualquier tipo puede aparecer dentro de las llaves, y C# convertirá la expresión en una cadena llamando a su método ToString o equivalente. Puedes cambiar el formato agregando a la expresión dos puntos y una cadena de formato (los formatos de cadena se describen en "String.Format y cadenas de formato compuestas" en la página 276):

```
string s = $"255 in hex is {byte.MaxValue:X2}";  // X2 = 2-digit hexadecimal
// Evaluates to "255 in hex is FF"
```

Si necesitas utilizar dos puntos para otro propósito (como un operador condicional ternario, que se explicará más adelante), debes envolver toda la expresión entre paréntesis:

```
bool b = true;
Console.WriteLine ($"The answer in binary is {(b ? 1 : 0)}");
```

Las cadenas interpoladas deben completarse en una sola línea, a menos que también especifiques el operador de cadena de texto con formato:

```
int x = 2;
// Note that $ must appear before @ prior to C# 8:
string s = $@"this interpolation spans {
x} lines";
```

Para incluir un carácter de llave literal en una cadena interpolada, repite el carácter de llave deseado.

Comparaciones de cadenas

string no admite los operadores < y > para comparaciones. Debes usar el método CompareTo de la cadena, que se describe en el Capítulo 6.

Cadenas interpoladas constantes (C# 10)

A partir de C# 10, las cadenas interpoladas pueden ser constantes, siempre y cuando los valores interpolados también sean constantes:

```
const string greeting = "Hello";
const string message = $"{greeting}, world";
```

Arreglos

Un arreglo representa un número fijo de variables (llamadas elementos) de un tipo específico. Los elementos de un arreglo siempre se almacenan en un bloque contiguo de memoria, lo que proporciona un acceso altamente eficiente.

Un arreglo se denota con corchetes cuadrados después del tipo de elemento:

```
char[] vowels = new char[5];    // Declare an array of 5 characters
```

Los corchetes cuadrados también se utilizan para indexar el arreglo, accediendo a un elemento específico por su posición:

```
vowels[0] = 'a';
vowels[1] = 'e';
vowels[2] = 'i';
vowels[3] = 'o';
vowels[4] = 'u';
Console.WriteLine (vowels[1]);      // e
```

Esto imprime "e" porque los índices del arreglo comienzan en 0. Puedes utilizar un bucle for para iterar a través de cada elemento del arreglo. El bucle for en este ejemplo recorre el entero i desde 0 hasta 4:

```
for (int i = 0; i < vowels.Length; i++)
  Console.Write (vowels[i]);            // aeiou
```

La propiedad Length de un arreglo devuelve el número de elementos en el arreglo. Una vez que se ha creado un arreglo, no puedes cambiar su longitud. El espacio de nombres System.Collection y sus subespacios de nombres proporcionan estructuras de datos de nivel superior, como arreglos de tamaño dinámico y diccionarios.

Una expresión de inicialización de arreglo te permite declarar y poblar un arreglo en un solo paso:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

O simplemente:

```
char[] vowels = {'a','e','i','o','u'};
```

Todos los arreglos heredan de la clase System.Array, lo que proporciona servicios comunes para todos los arreglos. Estos miembros incluyen métodos para obtener y establecer elementos independientemente del tipo de arreglo. Se describen en "La clase Array" en la página 355.

Inicialización de elementos predeterminados

Crear un arreglo siempre inicializa los elementos con valores predeterminados. El valor predeterminado para un tipo es el resultado de un borrado de bits en la memoria. Por ejemplo, considera la creación de un arreglo de enteros. Debido a que int es un tipo de valor, esto asigna 1,000 enteros en un bloque de memoria contiguo. El valor predeterminado para cada elemento será 0:

```
int[] a = new int[1000];
Console.Write (a[123]);           // 0
```

Tipos de valor versus tipos de referencia

Si el tipo de elemento de un arreglo es un tipo de valor o un tipo de referencia tiene importantes implicaciones de rendimiento. Cuando el tipo de elemento es un tipo de valor, cada valor de elemento se asigna como parte del arreglo, como se muestra aquí:

```
Point[] a = new Point[1000];
int x = a[500].X;                 // 0

public struct Point { public int X, Y; }
```

Si Point hubiera sido una clase, crear el arreglo simplemente habría asignado 1,000 referencias nulas:

```
Point[] a = new Point[1000];
int x = a[500].X;                 // Runtime error, NullReferenceException

public class Point { public int X, Y; }
```

Para evitar este error, debemos instanciar explícitamente 1,000 Points después de instanciar el arreglo:

```
Point[] a = new Point[1000];
for (int i = 0; i < a.Length; i++) // Iterate i from 0 to 999
   a[i] = new Point();             // Set array element i with new point
```

Un arreglo en sí siempre es un objeto de tipo de referencia, independientemente del tipo de elemento. Por ejemplo, lo siguiente es válido:

```
int[] a = null;
```

Índices y rangos

Los índices y rangos (introducidos en C# 8) simplifican el trabajo con elementos o porciones de un arreglo.



Los índices y rangos también funcionan con los tipos CLR Span<T> y ReadOnlySpan<T> (consultar Capítulo 23).

También puedes hacer que tus propios tipos funcionen con índices y rangos, definiendo un indizador de tipo Index o Range (consultar "Indizadores" en la página 109).

Índices

Los índices te permiten hacer referencia a elementos en relación al final de un arreglo, utilizando el operador ^. ^1 se refiere al último elemento, ^2 se refiere al penúltimo elemento, y así sucesivamente:

```
char[] vowels = new char[] {'a','e','i','o','u'};
char lastElement  = vowels [^1];   // 'u'
char secondToLast = vowels [^2];   // 'o'
```

(^0 equivale a la longitud del arreglo, por lo que vowels[^0] genera un error).

C# implementa los índices con la ayuda del tipo Index, por lo que también puedes hacer lo siguiente:

```
Index first = 0;
Index last = ^1;
char firstElement = vowels [first];   // 'a'
char lastElement = vowels [last];     // 'u'
```

Rangos

Los rangos te permiten "rebanar" un arreglo utilizando el operador .. :

```
char[] firstTwo =  vowels [..2];    // 'a', 'e'
char[] lastThree = vowels [2..];    // 'i', 'o', 'u'
char[] middleOne = vowels [2..3];   // 'i'
```

El segundo número en el rango es exclusivo, por lo que ..2 devuelve los elementos antes de vowels[2].

También puedes usar el símbolo ^ en los rangos. Lo siguiente devuelve los últimos dos caracteres:

```
char[] lastTwo = vowels [^2..];     // 'o', 'u'
```

C# implementa los rangos con la ayuda del tipo Range, por lo que también puedes hacer lo siguiente:

```
Range firstTwoRange = 0..2;
char[] firstTwo = vowels [firstTwoRange];   // 'a', 'e'
```

Arreglos Multidimensionales

Los arreglos multidimensionales vienen en dos variedades: rectangulares y escalonados (jagged). Los arreglos rectangulares representan un bloque de memoria n-dimensional, y los arreglos escalonados son arreglos de arreglos.

Arreglos rectangulares

Los arreglos rectangulares se declaran utilizando comas para separar cada dimensión. El siguiente ejemplo declara un arreglo bidimensional rectangular con dimensiones de 3 por 3:

```
int[,] matrix = new int[3,3];
```

El método GetLength de un arreglo devuelve la longitud para una dimensión específica (comenzando en 0):

```
for (int i = 0; i < matrix.GetLength(0); i++)
  for (int j = 0; j < matrix.GetLength(1); j++)
    matrix[i,j] = i * 3 + j;
```

Puedes inicializar un arreglo rectangular con valores explícitos. El siguiente código crea un arreglo idéntico al ejemplo anterior:

```
int[,] matrix = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
```

Arreglos escalonados

Los arreglos escalonados se declaran utilizando corchetes sucesivos para representar cada dimensión. Aquí tienes un ejemplo de declaración de un arreglo bidimensional escalonado donde la dimensión externa es 3:

```
int[][] matrix = new int[3][];
```



Interesantemente, esto es new int[3][] y no new int[][3]. Eric Lippert ha escrito un excelente artículo sobre por qué es así.

Las dimensiones internas no se especifican en la declaración porque, a diferencia de una matriz rectangular, cada matriz interna puede tener una longitud arbitraria. Cada matriz interna se inicializa implícitamente como nula en lugar de como una matriz vacía. Debes crear manualmente cada matriz interna:

```
for (int i = 0; i < matrix.Length; i++)
{
  matrix[i] = new int[3];                    // Create inner array
  for (int j = 0; j < matrix[i].Length; j++)
    matrix[i][j] = i * 3 + j;
}
```

También puedes inicializar una matriz irregular con valores explícitos. El siguiente código crea una matriz idéntica al ejemplo anterior con un elemento adicional al final:

```
int[][] matrix = new int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8,9}
};
```

Expresiones Simplificadas de Inicialización de Matrices

Hay dos formas de acortar las expresiones de inicialización de matrices. La primera es omitir el operador new y las calificaciones de tipo:

```
char[] vowels = {'a','e','i','o','u'};

int[,] rectangularMatrix =
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};

int[][] jaggedMatrix =
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8,9}
};
```

El segundo enfoque es utilizar la palabra clave "var", que indica al compilador que infiera automáticamente el tipo de una variable local:

```
var i = 3;          // i is implicitly of type int
var s = "sausage";  // s is implicitly of type string

// Therefore:

var rectMatrix = new int[,]    // rectMatrix is implicitly of type int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};

var jaggedMat = new int[][]    // jaggedMat is implicitly of type int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8,9}
};
```

La tipificación implícita se puede llevar un paso más allá con los arreglos: puedes omitir el calificador de tipo después de la palabra clave "new" y hacer que el compilador infiera el tipo del arreglo:

```
var vowels = new[] {'a','e','i','o','u'};   // Compiler infers char[]
```

Para que esto funcione, todos los elementos deben ser implícitamente convertibles a un solo tipo (y al menos uno de los elementos debe ser de ese tipo, y debe haber exactamente un mejor tipo), como en el siguiente ejemplo:

```
var x = new[] {1,10000000000};   // all convertible to long
```

Verificación de límites

Todas las indexaciones de arreglos son verificadas por límites por el tiempo de ejecución. Se lanza una excepción "IndexOutOfRangeException" si se utiliza un índice no válido:

```
int[] arr = new int[3];
arr[3] = 1;               // IndexOutOfRangeException thrown
```

La verificación de límites de matriz es necesaria para la seguridad del tipo y simplifica la depuración.



Generalmente, el impacto en el rendimiento debido a la verificación de límites es mínimo, y el compilador Just-in-Time (JIT) puede realizar optimizaciones, como determinar de antemano si todos los índices serán seguros antes de ingresar a un bucle, evitando así una verificación en cada iteración. Además, C# proporciona código "inseguro" que puede omitir explícitamente la verificación de límites (consulte "Código inseguro y punteros" en la página 243).

Variables y Parámetros

Una variable representa una ubicación de almacenamiento que tiene un valor modificable. Una variable puede ser una variable local, un parámetro (valor, ref, out o in), un campo (de instancia o estático) o un elemento de un arreglo.

La Pila y el Montón (Heap)

La pila y el montón son los lugares donde residen las variables. Cada uno tiene una semántica de tiempo de vida muy diferente.

Pila (Stack)

La pila es un bloque de memoria para almacenar variables locales y parámetros. La pila crece y disminuye de forma lógica a medida que se ingresa y se sale de un método o función. Considera el siguiente método (para evitar distracciones, se ignora la verificación de argumentos de entrada):

```
static int Factorial (int x)
{
  if (x == 0) return 1;
  return x * Factorial (x-1);
}
```

Este método es recursivo, lo que significa que se llama a sí mismo. Cada vez que se ingresa al método, se asigna un nuevo entero (int) en la pila, y cada vez que se sale del método, el entero se desasigna.

Montón (Heap)

El montón es la memoria en la que residen los objetos (instancias de tipos de referencia). Cada vez que se crea un nuevo objeto, se asigna en el montón y se devuelve una referencia a ese objeto. Durante la ejecución de un programa, el montón se va llenando a medida que se crean nuevos objetos. El tiempo de ejecución tiene un recolector de basura (garbage collector) que periódicamente desasigna objetos del montón para que el programa no se quede sin memoria. Un objeto es elegible para desasignación tan pronto como no sea referenciado por nada que esté "vivo" en sí mismo.

En el siguiente ejemplo, comenzamos creando un objeto StringBuilder referenciado por la variable ref1 y luego escribimos su contenido. Ese objeto StringBuilder es inmediatamente elegible para la recolección de basura porque nada más lo utiliza posteriormente.

Luego, creamos otro objeto StringBuilder referenciado por la variable ref2 y copiamos esa referencia en ref3. Aunque ref2 no se utiliza después de ese punto, ref3 mantiene vivo el mismo objeto StringBuilder, asegurando que no sea elegible para la recolección hasta que hayamos terminado de usar ref3:

```
using System;
using System.Text;

StringBuilder ref1 = new StringBuilder ("object1");
Console.WriteLine (ref1);
// The StringBuilder referenced by ref1 is now eligible for GC.

StringBuilder ref2 = new StringBuilder ("object2");
StringBuilder ref3 = ref2;
// The StringBuilder referenced by ref2 is NOT yet eligible for GC.

Console.WriteLine (ref3);      // object2
```

Las instancias de tipos de valor (y las referencias de objetos) viven donde se declaró la variable. Si la instancia se declaró como un campo dentro de un tipo de clase, o como un elemento de un arreglo, esa instancia vive en el montón (heap).



En C#, no se puede eliminar objetos de forma explícita, como se puede hacer en C++. Un objeto no referenciado es eventualmente recolectado por el recolector de basura (garbage collector).

El montón también almacena campos estáticos. A diferencia de los objetos asignados en el montón (que pueden ser recolectados por el recolector de basura), estos campos están presentes hasta que el proceso termine.

Asignación Definida

C# aplica una política de asignación definida. En la práctica, esto significa que, fuera de un contexto inseguro o de interoperabilidad, no puedes acceder accidentalmente a memoria no inicializada.

La asignación definida tiene tres implicaciones:

• Las variables locales deben recibir un valor antes de que puedan ser leídas.

• Los argumentos de una función deben ser proporcionados al llamar a un método (a menos que estén marcados como opcionales; consulta "Parámetros opcionales" en la página 65).

• Todas las demás variables (como campos y elementos de un arreglo) se inicializan automáticamente por el tiempo de ejecución.

Por ejemplo, el siguiente código genera un error en tiempo de compilación:

```
int x;
Console.WriteLine (x);        // Compile-time error
```

Los campos y elementos de un arreglo se inicializan automáticamente con los valores predeterminados de su tipo. El siguiente código muestra 0 porque los elementos del arreglo se asignan implícitamente a sus valores predeterminados:

```
int[] ints = new int[2];
Console.WriteLine (ints[0]);    // 0
```

El siguiente código muestra 0, porque los campos se asignan implícitamente a un valor predeterminado (ya sea de instancia o estático):

```
Console.WriteLine (Test.X);   // 0

class Test { public static int X; }   // field
```

Valores Predeterminados

Todas las instancias de tipos tienen un valor predeterminado. El valor predeterminado para los tipos predefinidos es el resultado de un borrado a cero bitwise de la memoria:

Type	Default value
Reference types (and nullable value types)	`null`
Numeric and enum types	`0`
`char` type	`'\0'`
`bool` type	`false`

Puedes obtener el valor predeterminado para cualquier tipo utilizando la palabra clave "default":

```
Console.WriteLine (default (decimal));   // 0
```

Opcionalmente, puedes omitir el tipo cuando se puede inferir:

```
decimal d = default;
```

El valor predeterminado en un tipo de valor personalizado (es decir, una estructura) es el mismo que el valor predeterminado para cada campo definido por el tipo personalizado.

Parámetros

Un método puede tener una secuencia de parámetros. Los parámetros definen el conjunto de argumentos que deben proporcionarse para ese método. En el siguiente ejemplo, el método "Foo" tiene un único parámetro llamado "p", de tipo entero:

```
Foo (8);                        // 8 is an argument
static void Foo (int p) {...}   // p is a parameter
```

Puedes controlar cómo se pasan los parámetros con los modificadores "ref", "in" y "out":

Parameter modifier	Passed by	Variable must be definitely assigned
(None)	Value	Going in
`ref`	Reference	Going in
`in`	Reference (read-only)	Going in
`out`	Reference	Going out

Pasar argumentos por valor

De forma predeterminada, los argumentos en C# se pasan por valor, que es el caso más común. Esto significa que se crea una copia del valor cuando se pasa al método:

```
int x = 8;
Foo (x);                    // Make a copy of x
Console.WriteLine (x);      // x will still be 8

static void Foo (int p)
{
  p = p + 1;                // Increment p by 1
  Console.WriteLine (p);    // Write p to screen
}
```

Asignar un nuevo valor a p no cambia el contenido de x, porque p y x residen en ubicaciones de memoria diferentes.

Pasar un argumento de tipo de referencia por valor copia la referencia pero no el objeto en sí. En el siguiente ejemplo, Foo ve el mismo objeto StringBuilder que hemos instanciado (sb), pero tiene una referencia independiente a él. En otras palabras, sb y fooSB son variables separadas que hacen referencia al mismo objeto StringBuilder:

```
StringBuilder sb = new StringBuilder();
Foo (sb);
Console.WriteLine (sb.ToString());    // test

static void Foo (StringBuilder fooSB)
{
  fooSB.Append ("test");
  fooSB = null;
}
```

Debido a que fooSB es una copia de una referencia, establecerlo como null no hace que sb sea null. (Sin embargo, si fooSB se declarara y se llamara con el modificador ref, sb se volvería null).

El modificador ref

Para pasar por referencia, C# proporciona el modificador de parámetro ref. En el siguiente ejemplo, p y x se refieren a las mismas ubicaciones de memoria:

```
int x = 8;
Foo (ref  x);              // Ask Foo to deal directly with x
Console.WriteLine (x);     // x is now 9

static void Foo (ref int p)
{
  p = p + 1;               // Increment p by 1
  Console.WriteLine (p);   // Write p to screen
}
```

Ahora, asignarle a p un nuevo valor cambia el contenido de x. Observa cómo se requiere el modificador ref tanto al escribir como al llamar al método. Esto hace que quede muy claro lo que está sucediendo.

El modificador ref es esencial para implementar un método de intercambio (en "Genéricos" en la página 145, mostramos cómo escribir un método de intercambio que funciona con cualquier tipo):

```
string x = "Penn";
string y = "Teller";
Swap (ref x, ref y);
Console.WriteLine (x);   // Teller
Console.WriteLine (y);   // Penn

static void Swap (ref string a, ref string b)
{
  string temp = a;
  a = b;
  b = temp;
}
```


Un parámetro puede ser pasado por referencia o por valor, independientemente de si el tipo de parámetro es un tipo de referencia o un tipo de valor.

El modificador "out"

Un argumento "out" es similar a un argumento "ref", excepto por lo siguiente:

- No es necesario asignarlo antes de entrar en la función.
- Debe ser asignado antes de salir de la función.

El modificador "out" se utiliza comúnmente para obtener múltiples valores de retorno de un método; por ejemplo:

```
string a, b;
Split ("Stevie Ray Vaughn", out a, out b);
Console.WriteLine (a);                      // Stevie Ray
Console.WriteLine (b);                      // Vaughn

void Split (string name, out string firstNames, out string lastName)
{
  int i = name.LastIndexOf (' ');
  firstNames = name.Substring (0, i);
  lastName = name.Substring (i + 1);
}
```

Al igual que un parámetro "ref", un parámetro "out" se pasa por referencia.

Variables y descartes con "out"

Puedes declarar variables sobre la marcha al llamar a métodos con parámetros "out". Podemos reemplazar las dos primeras líneas de nuestro ejemplo anterior con esto:

```
Split ("Stevie Ray Vaughan", out string a, out string b);
```

Cuando llamas a métodos con múltiples parámetros "out", a veces no estás interesado en recibir valores de todos los parámetros. En esos casos, puedes "descartar" aquellos en los que no estás interesado utilizando un guion bajo:

```
Split ("Stevie Ray Vaughan", out string a, out _);   // Discard 2nd param
Console.WriteLine (a);
```

En este caso, el compilador trata el guion bajo como un símbolo especial, llamado descarte. Puedes incluir múltiples descartes en una sola llamada. Suponiendo que SomeBigMethod haya sido definido con siete parámetros "out", podemos ignorar todos excepto el cuarto de la siguiente manera:

```
SomeBigMethod (out _, out _, out _, out int x, out _, out _, out _);
```

Por razones de compatibilidad con versiones anteriores, esta característica del lenguaje no surtirá efecto si existe una variable guion bajo real en el ámbito:

```
string _;
Split ("Stevie Ray Vaughan", out string a, out _);
Console.WriteLine (_);     // Vaughan
```

Implicaciones de pasar por referencia

Cuando pasas un argumento por referencia, estás creando un alias a la ubicación de almacenamiento de una variable existente en lugar de crear una nueva ubicación de almacenamiento. En el siguiente ejemplo, las variables x e y representan la misma instancia:

```
class Test
{
  static int x;

  static void Main() { Foo (out x); }

  static void Foo (out int y)
  {
    Console.WriteLine (x);                // x is 0
    y = 1;                                // Mutate y
    Console.WriteLine (x);                // x is 1
  }
}
```

El modificador "in"

Un parámetro "in" es similar a un parámetro "ref", excepto que el valor del argumento no puede ser modificado por el método (hacerlo generaría un error en tiempo de compilación). Este modificador es especialmente útil cuando se pasa un tipo de valor grande al método, ya que permite al compilador evitar la sobrecarga de copiar el argumento antes de pasarlo, al mismo tiempo que protege el valor original de modificaciones.

Sobrecargar únicamente en función de la presencia de `in` está permitido:

```
void Foo (   SomeBigStruct a) { ... }
void Foo (in SomeBigStruct a) { ... }
```

Para llamar a la segunda sobrecarga, el llamador debe utilizar el modificador `in`:

```
SomeBigStruct x = ...;
Foo (x);      // Calls the first overload
Foo (in x);   // Calls the second overload
```

Cuando no hay ambigüedad,

```
void Bar (in SomeBigStruct a) { ... }
```

el uso del modificador `in` es opcional para el llamador:

```
Bar (x);     // OK (calls the 'in' overload)
Bar (in x);  // OK (calls the 'in' overload)
```

Para que este ejemplo sea significativo, `SomeBigStruct` debería estar definido como una estructura (ver "Estructuras" en la página 129).

El modificador `params`

El modificador `params`, si se aplica al último parámetro de un método, permite que el método acepte cualquier número de argumentos de un tipo particular. El tipo del parámetro debe declararse como un array unidimensional, como se muestra en el siguiente ejemplo:

```
int total = Sum (1, 2, 3, 4);
Console.WriteLine (total);              // 10

// The call to Sum above is equivalent to:
int total2 = Sum (new int[] { 1, 2, 3, 4 });

int Sum (params int[] ints)
{
  int sum = 0;
  for (int i = 0; i < ints.Length; i++)
    sum += ints [i];                        // Increase sum by ints[i]
  return sum;
}
```

Si no hay argumentos en la posición `params`, se crea un array de longitud cero. También puedes suministrar un argumento `params` como un array normal. La primera línea de nuestro ejemplo es semánticamente equivalente a esto:

```
int total = Sum (new int[] { 1, 2, 3, 4 } );
```

Parámetros opcionales

Los métodos, constructores e indexadores (Capítulo 3) pueden declarar parámetros opcionales. Un parámetro es opcional si especifica un valor predeterminado en su declaración:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

Puedes omitir los parámetros opcionales al llamar al método:

```
Foo();     // 23
```

El argumento predeterminado de 23 se pasa en realidad al parámetro opcional `x`; el compilador incrusta el valor 23 en el código compilado en el lado de la llamada. La llamada anterior a `Foo` es semánticamente idéntica a:

```
Foo (23);
```

porque el compilador simplemente sustituye el valor predeterminado de un parámetro opcional donde se utiliza.

 Agregar un parámetro opcional a un método público que se llama desde otro ensamblado requiere la recompilación de ambos ensamblados, al igual que si el parámetro fuera obligatorio.

El valor predeterminado de un parámetro opcional debe ser especificado por una expresión constante, un constructor sin parámetros de un tipo de valor o una expresión predeterminada. Los parámetros opcionales no pueden ser marcados con ref o out.

Los parámetros obligatorios deben aparecer antes de los parámetros opcionales tanto en la declaración del método como en la llamada al método (la excepción son los argumentos params, que siempre deben ir al final). En el siguiente ejemplo, se pasa el valor explícito 1 a x y el valor predeterminado 0 a y:

```
Foo (1);    // 1, 0

void Foo (int x = 0, int y = 0) { Console.WriteLine (x + ", " + y); }
```

Puedes hacer lo contrario (pasar un valor predeterminado a x y un valor explícito a y) combinando parámetros opcionales con argumentos nombrados.

Argumentos nombrados

En lugar de identificar un argumento por posición, puedes identificarlo por nombre:

```
Foo (x:1, y:2);  // 1, 2

void Foo (int x, int y) { Console.WriteLine (x + ", " + y); }
```

Los argumentos nombrados pueden aparecer en cualquier orden. Las siguientes llamadas a Foo son semánticamente idénticas:

```
Foo (x:1, y:2);
Foo (y:2, x:1);
```



Una diferencia sutil es que las expresiones de los argumentos se evalúan en el orden en que aparecen en el sitio de llamada. En general, esto solo marca una diferencia con expresiones interdependientes que generan efectos secundarios, como el siguiente ejemplo, que escribe 0, 1:

```
int a = 0;
Foo (y: ++a, x: --a);  // ++a is evaluated first
```

¡Por supuesto! Casi con seguridad evitarías escribir ese tipo de código en la práctica.

Puedes mezclar argumentos posicionales y nombrados de la siguiente manera:

```
Foo (1, y:2);
```

Sin embargo, hay una restricción: los argumentos posicionales deben venir antes de los argumentos nombrados, a menos que se utilicen en la posición correcta. Entonces, podrías llamar a Foo de la siguiente manera:

```
Foo (x:1, 2);         // OK. Arguments in the declared positions
```

Pero no de esta manera:

```
Foo (y:2, 1);         // Compile-time error. y isn't in the first position
```

Los argumentos nombrados son particularmente útiles en conjunción con parámetros opcionales. Por ejemplo, considera el siguiente método:

```
void Bar (int a = 0, int b = 0, int c = 0, int d = 0) { ... }
```

Puedes llamar a este método suministrando solo un valor para d, de la siguiente manera:

```
Bar (d:3);
```

Esto es especialmente útil al llamar a APIs COM, las cuales se discuten en detalle en el Capítulo 24.

Referencias locales

Una característica algo esotérica de C# es que puedes definir una variable local que haga referencia a un elemento en un array o campo de un objeto (a partir de C# 7):

```
int[] numbers = { 0, 1, 2, 3, 4 };
ref int numRef = ref numbers [2];
```

En este ejemplo, numRef es una referencia a numbers[2]. Cuando modificamos numRef, modificamos el elemento del array:

```
numRef *= 10;
Console.WriteLine (numRef);        // 20
Console.WriteLine (numbers [2]);   // 20
```

El destino de una referencia local (ref local) debe ser un elemento de un array, un campo o una variable local; no puede ser una propiedad (Capítulo 3). Las referencias locales se utilizan en escenarios de micro-optimización especializados y normalmente se usan en conjunto con retornos por referencia.

Retornos por referencia



Los tipos Span<T> y ReadOnlySpan<T> que describimos en el Capítulo 23 utilizan retornos por referencia (ref returns) para implementar un indexador altamente eficiente. Fuera de esos escenarios, los retornos por referencia no se utilizan comúnmente y puedes considerarlos como una característica de micro-optimización.

Puedes devolver una referencia local desde un método. Esto se llama "retorno por referencia" (ref return):

```
class Program
{
  static string x = "Old Value";

  static ref string GetX() => ref x;     // This method returns a ref

  static void Main()
  {
    ref string xRef = ref GetX();        // Assign result to a ref local
    xRef = "New Value";
    Console.WriteLine (x);               // New Value
  }
}
```

Si omites el modificador ref en el lado de la llamada, se devuelve un valor ordinario:

```
string localX = GetX();  // Legal: localX is an ordinary non-ref variable.
```

También puedes utilizar retornos por referencia al definir una propiedad o un indexador:

```
static ref string Prop => ref x;
```

Una propiedad de este tipo es implícitamente modificable, a pesar de no tener un accesor set:

```
Prop = "New Value";
```

Puedes evitar dicha modificación utilizando ref readonly:

```
static ref readonly string Prop => ref x;
```

El modificador ref readonly evita la modificación pero aún permite obtener la ganancia de rendimiento al devolver por referencia. La ganancia sería muy pequeña en este caso, ya que x es de tipo string (un tipo de referencia): sin importar cuán larga sea la cadena, la única ineficiencia que puedes evitar es la copia de una sola referencia de 32 o 64 bits. Se pueden obtener ganancias reales con tipos de valor personalizados (ver "Structs" en la página 129), pero solo si la estructura está marcada como readonly (de lo contrario, el compilador realizará una copia defensiva).

Intentar definir un accesor set explícito en una propiedad o un indexador que devuelve una referencia (ref) es ilegal.

var: Variables de tipo implícito

Es frecuente que declares e inicialices una variable en un solo paso. Si el compilador puede inferir el tipo a partir de la expresión de inicialización, puedes utilizar la palabra clave var en lugar de la declaración de tipo; por ejemplo:

```
var x = "hello";
var y = new System.Text.StringBuilder();
var z = (float)Math.PI;
```

Esto es precisamente equivalente a lo siguiente:

```
string x = "hello";
System.Text.StringBuilder y = new System.Text.StringBuilder();
float z = (float)Math.PI;
```

Debido a esta equivalencia directa, las variables de tipo implícito son estáticamente tipadas. Por ejemplo, lo siguiente generaría un error en tiempo de compilación:

```
var x = 5;
x = "hello";    // Compile-time error; x is of type int
```



var puede disminuir la legibilidad del código cuando no se puede deducir el tipo únicamente al mirar la declaración de la variable; por ejemplo:

```
Random r = new Random();
var x = r.Next();
```

¿Qué tipo es x?

En "Tipos anónimos" en la página 205, describiremos un escenario en el cual el uso de var es obligatorio.

Expresiones new con objetivo tipado

Otra forma de reducir la repetición léxica es mediante las expresiones new con objetivo tipado (a partir de C# 9):

```
System.Text.StringBuilder sb1 = new();
System.Text.StringBuilder sb2 = new ("Test");
```

Esto es precisamente equivalente a:

```
System.Text.StringBuilder sb1 = new System.Text.StringBuilder();
System.Text.StringBuilder sb2 = new System.Text.StringBuilder ("Test");
```

El principio es que puedes llamar a new sin especificar el nombre del tipo si el compilador es capaz de inferirlo sin ambigüedad. Las expresiones new con objetivo tipado son especialmente útiles cuando la declaración y la inicialización de la variable están en diferentes partes de tu código. Un ejemplo común es cuando deseas inicializar un campo en un constructor:

```
class Foo
{
  System.Text.StringBuilder sb;

  public Foo (string initialValue)
  {


      sb = new (initialValue);
    }
  }
```

Las expresiones new con objetivo tipado también son útiles en el siguiente escenario:

```
MyMethod (new ("test"));

void MyMethod (System.Text.StringBuilder sb) { ... }
```

Expresiones y operadores

Una expresión denota principalmente un valor. Los tipos más simples de expresiones son las constantes y las variables. Las expresiones se pueden transformar y combinar utilizando operadores. Un operador toma uno o más operandos de entrada para producir una nueva expresión. Aquí tienes un ejemplo de una expresión constante:

```
12
```

Podemos utilizar el operador * para combinar dos operandos (las expresiones literales 12 y 30), de la siguiente manera:

```
12 * 30
```

Podemos construir expresiones complejas porque un operando puede ser en sí mismo una expresión, como el operando (12 * 30) en el siguiente ejemplo:

```
1 + (12 * 30)
```

Los operadores en C# pueden ser clasificados como unarios, binarios o ternarios, dependiendo del número de operandos en los que trabajan (uno, dos o tres). Los operadores binarios siempre utilizan la notación infija en la que el operador se coloca entre los dos operandos.

Expresiones primarias

Las expresiones primarias incluyen expresiones compuestas por operadores que son intrínsecos a los aspectos fundamentales del lenguaje. Aquí tienes un ejemplo:

```
Math.Log (1)
```

Esta expresión está compuesta por dos expresiones primarias. La primera expresión realiza una búsqueda de miembro (con el operador .), y la segunda expresión realiza una llamada a un método (con el operador ()).

Expresiones void

Una expresión void es una expresión que no tiene un valor, como esta:

```
Console.WriteLine (1)
```

Debido a que no tiene un valor, no puedes utilizar una expresión void como operando para construir expresiones más complejas:

```
1 + Console.WriteLine (1)      // Compile-time error
```

Expresiones de asignación

Una expresión de asignación utiliza el operador = para asignar el resultado de otra expresión a una variable. Por ejemplo:

```
x = x * 5
```

Una expresión de asignación no es una expresión void, tiene un valor que es lo que se asignó, por lo que puede ser incorporada en otra expresión. En el siguiente ejemplo, la expresión asigna 2 a x y 10 a y:

```
y = 5 * (x = 2)
```

Puedes usar este estilo de expresión para inicializar múltiples valores:

```
a = b = c = d = 0
```

Los operadores de asignación compuesta son atajos sintácticos que combinan la asignación con otro operador:

```
x *= 2    // equivalent to x = x * 2
x <<= 1   // equivalent to x = x << 1
```

(Una excepción sutil a esta regla ocurre con los eventos, que se describen en el Capítulo 4: los operadores += y -= aquí se tratan de manera especial y se mapean a los accesores add y remove del evento).

Precedencia de operadores y asociatividad

Cuando una expresión contiene múltiples operadores, la precedencia y la asociatividad determinan el orden de evaluación. Los operadores con mayor precedencia se ejecutan antes que los operadores con menor precedencia. Si los operadores tienen la misma precedencia, la asociatividad del operador determina el orden de evaluación.

Precedencia

La siguiente expresión

```
1 + 2 * 3
```

se evalúa de la siguiente manera porque * tiene una precedencia mayor que +:

```
1 + (2 * 3)
```

Operadores asociativos a la izquierda

Los operadores binarios (excepto los operadores de asignación, lambda y null-coalescing) son asociativos a la izquierda, es decir, se evalúan de izquierda a derecha. Por ejemplo, la siguiente expresión

```
8 / 4 / 2
```

se evalúa de la siguiente manera:

```
( 8 / 4 ) / 2    // 1
```

Puedes insertar paréntesis para cambiar el orden real de evaluación:

```
8 / ( 4 / 2 )    // 4
```

Operadores asociativos a la derecha

Los operadores de asignación, así como los operadores lambda, null-coalescing y condicionales, son asociativos a la derecha, es decir, se evalúan de derecha a izquierda. La asociatividad a la derecha permite múltiples asignaciones, como la siguiente, para compilar:

```
x = y = 3;
```

Esto asigna primero 3 a y y luego asigna el resultado de esa expresión (3) a x.

Tabla de operadores

La Tabla 2-3 lista los operadores de C# en orden de precedencia. Los operadores en la misma categoría tienen la misma precedencia. Explicamos los operadores sobrecargables por el usuario en "Sobrecarga de operadores" en la página 239.

Table 2-3. C# operators (categories in order of precedence)

Category	Operator symbol	Operator name	Example	User-overloadable
Primary	`.`	Member access	`x.y`	No
	`?.` and `?[]`	Null-conditional	`x?.y` or `x?[0]`	No
	`!` (postfix)	Null-forgiving	`x!.y` or `x![0]`	No
	`->` (unsafe)	Pointer to struct	`x->y`	No
	`()`	Function call	`x()`	No
	`[]`	Array/index	`a[x]`	Via indexer
	`++`	Post-increment	`x++`	Yes
	`--`	Post-decrement	`x--`	Yes
	`new`	Create instance	`new Foo()`	No
	`stackalloc`	Stack allocation	`stackalloc(10)`	No
	`typeof`	Get type from identifier	`typeof(int)`	No
	`nameof`	Get name of identifier	`nameof(x)`	No
	`checked`	Integral overflow check on	`checked(x)`	No
	`unchecked`	Integral overflow check off	`unchecked(x)`	No
	`default`	Default value	`default(char)`	No
Unary	`await`	Await	`await myTask`	No

Category	Operator symbol	Operator name	Example	User-overloadable
	`sizeof`	Get size of struct	`sizeof(int)`	No
	`+`	Positive value of	`+x`	Yes
	`-`	Negative value of	`-x`	Yes
	`!`	Not	`!x`	Yes
	`~`	Bitwise complement	`~x`	Yes
	`++`	Pre-increment	`++x`	Yes
	`--`	Pre-decrement	`--x`	Yes
	`()`	Cast	`(int)x`	No
	`^`	Index from end	`array[^1]`	No
	`*` (unsafe)	Value at address	`*x`	No
	`&` (unsafe)	Address of value	`&x`	No
Range	`..` `..^`	Range of indices	`x..y` `x..^y`	No

Switch & with	`switch`	Switch expression	`num switch {`
			`1 => true,`
			`_ => false`
			`}`
				No

	`with`	With expression	`rec with` `{ X = 123 }`	No
Multiplicative	`*`	Multiply	`x * y`	Yes
	`/`	Divide	`x / y`	Yes
	`%`	Remainder	`x % y`	Yes
Additive	`+`	Add	`x + y`	Yes
	`-`	Subtract	`x - y`	Yes
Shift	`<<`	Shift left	`x << 1`	Yes
	`>>`	Shift right	`x >> 1`	Yes
Relational	`<`	Less than	`x < y`	Yes
	`>`	Greater than	`x > y`	Yes
	`<=`	Less than or equal to	`x <= y`	Yes
	`>=`	Greater than or equal to	`x >= y`	Yes
	`is`	Type is or is subclass of	`x is y`	No
	`as`	Type conversion	`x as y`	No
Equality	`==`	Equals	`x == y`	Yes

Category	Operator symbol	Operator name	Example	User-overloadable
	`!=`	Not equals	`x != y`	Yes
Logical And	`&`	And	`x & y`	Yes
Logical Xor	`^`	Exclusive Or	`x ^ y`	Yes
Logical Or	`|`	Or	`x | y`	Yes
Conditional And	`&&`	Conditional And	`x && y`	Via &
Conditional Or	`||`	Conditional Or	`x || y`	Via |
Null coalescing	`??`	Null coalescing	`x ?? y`	No
Conditional	`?:`	Conditional	`isTrue ? thenThis` `: elseThis`	No
Assignment and lambda	`=`	Assign	`x = y`	No
	`*=`	Multiply self by	`x *= 2`	Via *
	`/=`	Divide self by	`x /= 2`	Via /
	`%=`	Remainder & assign to self	`x %= 2`	
	`+=`	Add to self	`x += 2`	Via +
	`-=`	Subtract from self	`x -= 2`	Via -
	`<<=`	Shift self left by	`x <<= 2`	Via <<
	`>>=`	Shift self right by	`x >>= 2`	Via >>
	`&=`	And self by	`x &= 2`	Via &
	`^=`	Exclusive-Or self by	`x ^= 2`	Via ^
	`|=`	Or self by	`x |= 2`	Via |
	`??=`	Null-coalescing assignment	`x ??= 0`	No
	`=>`	Lambda	`x => x + 1`	No

Operadores de nulos

C# proporciona tres operadores para facilitar el trabajo con valores nulos: el operador de fusión de nulos, el operador de asignación de fusión de nulos y el operador condicional de nulos.

Operador de Fusión de Nulos

El operador ?? es el operador de fusión de nulos. Indica: "Si el operando de la izquierda no es nulo, dámelo; de lo contrario, dame otro valor". Por ejemplo:

```
string s1 = null;
string s2 = s1 ?? "nothing";   // s2 evaluates to "nothing"
```

Si la expresión de la izquierda no es nula, la expresión de la derecha no se evalúa.

El operador de fusión nula también funciona con tipos de valor anulables (ver "Tipos de Valor Anulables" en la página 194).

Operador de asignación de fusión nula

El operador ??= (introducido en C# 8) es el operador de asignación de fusión nula. Dice: "Si el operando de la izquierda es nulo, asigna el operando de la derecha al operando de la izquierda".

Considera lo siguiente:

```
myVariable ??= someDefault;
```

Esto es equivalente a:

```
if (myVariable == null) myVariable = someDefault;
```

El operador ??= es especialmente útil en la implementación de propiedades calculadas de forma perezosa.

Abordaremos este tema más adelante, en "Campos Calculados y Evaluación Perezosa" en la página 217.

Operador nulo-condicional

El operador ?. es el operador nulo-condicional o "operador Elvis" (por el emoticón de Elvis). Te permite llamar métodos y acceder a miembros al igual que el operador de punto estándar, excepto que si el operando de la izquierda es nulo, la expresión se evalúa como nula en lugar de lanzar una excepción NullReferenceException.

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString();  // No error; s instead evaluates to null
```

La última línea es equivalente a lo siguiente:

```
string s = (sb == null ? null : sb.ToString());
```

Las expresiones nulo-condicionales también funcionan con indexadores:

```
string foo = null;
char? c = foo?[1];  // c is null
```

Al encontrar un valor nulo, el operador Elvis interrumpe la evaluación del resto de la expresión. En el siguiente ejemplo, s se evalúa como nulo, incluso con un operador de punto estándar entre ToString() y ToUpper():

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString().ToUpper();   // s evaluates to null without error
```

El uso repetido de Elvis es necesario solo si el operando inmediatamente a su izquierda puede ser nulo. La siguiente expresión es segura tanto si x es nulo como si x.y es nulo:

```
x?.y?.z
```

Es equivalente a lo siguiente (excepto que x.y se evalúa solo una vez):

```
x == null ? null
          : (x.y == null ? null : x.y.z)
```

La expresión final debe ser capaz de aceptar un valor nulo. Lo siguiente es ilegal:

```
System.Text.StringBuilder sb = null;
int length = sb?.ToString().Length;   // Illegal : int cannot be null
```

Podemos solucionar esto con el uso de tipos de valor anulables (ver "Tipos de Valor Anulables" en la página 194). Si ya estás familiarizado con los tipos de valor anulables, aquí tienes un adelanto:

```
int? length = sb?.ToString().Length;   // OK: int? can be null
```

También puedes usar el operador nulo-condicional para llamar a un método void:

```
someObject?.SomeVoidMethod();
```

Si someObject es nulo, esto se convierte en una "no operación" en lugar de lanzar una NullReferenceException.

Puedes utilizar el operador nulo-condicional con los miembros de tipo común que describimos en el Capítulo 3, incluyendo métodos, campos, propiedades e indexadores. También se combina bien con el operador de fusión nula:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString() ?? "nothing";   // s evaluates to "nothing"
```

Sentencias

Las funciones están compuestas por sentencias que se ejecutan secuencialmente en el orden textual en el que aparecen. Un bloque de sentencias es una serie de sentencias que se encuentran entre llaves (los tokens {}).

Sentencias de Declaración

Una declaración de variable introduce una nueva variable, opcionalmente inicializándola con una expresión. Puedes declarar múltiples variables del mismo tipo en una lista separada por comas:

```
string someWord = "rosebud";
int someNumber = 42;
bool rich = true, famous = false;
```

Una declaración de constante es similar a una declaración de variable, excepto que no se puede cambiar después de haber sido declarada y la inicialización debe ocurrir junto con la declaración (ver "Constantes" en la página 96):

```
const double c = 2.99792458E08;
c += 10;                        // Compile-time Error
```

Variables locales

El alcance de una variable local o constante local se extiende a lo largo del bloque actual. No puedes declarar otra variable local con el mismo nombre en el bloque actual o en bloques anidados:

```
int x;
{
  int y;
  int x;            // Error - x already defined


}
{
  int y;            // OK - y not in scope
}
Console.Write (y);  // Error - y is out of scope
```



El alcance de una variable se extiende en ambas direcciones a lo largo de su bloque de código. Esto significa que si moviéramos la declaración inicial de x en este ejemplo hacia la parte inferior del método, obtendríamos el mismo error. Esto contrasta con C++ y es algo peculiar, dado que no es legal hacer referencia a una variable o constante antes de que se haya declarado.

Declaraciones de Expresiones

Las declaraciones de expresiones son expresiones que también son válidas como declaraciones. Una declaración de expresión debe cambiar el estado o llamar a algo que pueda cambiar el estado. Cambiar el estado significa básicamente cambiar una variable. A continuación se muestran los posibles tipos de declaraciones de expresiones:

• Expresiones de asignación (incluyendo expresiones de incremento y decremento)

• Expresiones de llamada a métodos (tanto para métodos void como no void)

• Expresiones de instanciación de objetos

Aquí tienes algunos ejemplos:

```
// Declare variables with declaration statements:
string s;
int x, y;
System.Text.StringBuilder sb;

// Expression statements
x = 1 + 2;                // Assignment expression
x++;                      // Increment expression
y = Math.Max (x, 5);      // Assignment expression
Console.WriteLine (y);    // Method call expression
sb = new StringBuilder(); // Assignment expression
new StringBuilder();      // Object instantiation expression
```

Cuando llamas a un constructor o a un método que devuelve un valor, no estás obligado a utilizar el resultado. Sin embargo, a menos que el constructor o el método cambie el estado, la declaración es completamente inútil:

```
new StringBuilder();     // Legal, but useless
new string ('c', 3);     // Legal, but useless
x.Equals (y);            // Legal, but useless
```

Declaraciones de selección

C# tiene los siguientes mecanismos para controlar condicionalmente el flujo de ejecución del programa:

• Declaraciones de selección (if, switch)

• Operador condicional (?:)

• Declaraciones de bucle (while, do-while, for, foreach)

Esta sección cubre los dos constructos más simples: la declaración if y la declaración switch.

La declaración if

Una declaración if ejecuta una sentencia si una expresión bool es verdadera:

```
if (5 < 2 * 3)
  Console.WriteLine ("true");       // true
```

La sentencia puede ser un bloque de código:

```
if (5 < 2 * 3)
{
  Console.WriteLine ("true");
  Console.WriteLine ("Let's move on!");
}
```

La cláusula else

Una declaración if puede tener opcionalmente una cláusula else:

```
if (2 + 2 == 5)
  Console.WriteLine ("Does not compute");
else
  Console.WriteLine ("False");        // False
```

Dentro de una cláusula else, puedes anidar otra declaración if:

```
if (2 + 2 == 5)
  Console.WriteLine ("Does not compute");
else
  if (2 + 2 == 4)
    Console.WriteLine ("Computes");    // Computes
```

Cambiar el flujo de ejecución con llaves

Una cláusula else siempre se aplica a la declaración if inmediatamente anterior en el bloque de sentencias:

```
if (true)
  if (false)
    Console.WriteLine();


else
  Console.WriteLine ("executes");
```

Esto es semánticamente idéntico a lo siguiente:

```
if (true)
{
  if (false)
    Console.WriteLine();
  else
    Console.WriteLine ("executes");
}
```

Podemos cambiar el flujo de ejecución moviendo las llaves:

```
if (true)
{
  if (false)
    Console.WriteLine();
}
else
  Console.WriteLine ("does not execute");
```

Con las llaves, declaras explícitamente tu intención. Esto puede mejorar la legibilidad de las declaraciones if anidadas, incluso cuando no es requerido por el compilador. Una excepción notable es el siguiente patrón:

```
void TellMeWhatICanDo (int age)
{
  if (age >= 35)
    Console.WriteLine ("You can be president!");
  else if (age >= 21)
    Console.WriteLine ("You can drink!");
  else if (age >= 18)
    Console.WriteLine ("You can vote!");
  else
    Console.WriteLine ("You can wait!");
}
```

Aquí hemos organizado las declaraciones if y else para imitar la estructura "elseif" de otros lenguajes (y la directiva de preprocesador #elif de C#). El formateo automático de Visual Studio reconoce este patrón y mantiene la indentación. Sin embargo, semánticamente, cada declaración if que sigue a una declaración else está anidada funcionalmente dentro de la cláusula else.

La declaración switch

Las declaraciones switch te permiten ramificar la ejecución del programa basándote en una selección de posibles valores que una variable podría tener. Las declaraciones switch pueden resultar en un código más limpio que múltiples declaraciones if, porque las declaraciones switch requieren que se evalúe una expresión solo una vez:

```
void ShowCard (int cardNumber)
{
  switch (cardNumber)

  {
    case 13:
      Console.WriteLine ("King");
      break;
    case 12:
      Console.WriteLine ("Queen");
      break;
    case 11:
      Console.WriteLine ("Jack");
      break;
    case -1:                          // Joker is -1
      goto case 12;                   // In this game joker counts as queen
    default:                          // Executes for any other cardNumber
      Console.WriteLine (cardNumber);
      break;
  }
}
```

Este ejemplo demuestra el escenario más común, que es realizar un switch basado en constantes. Cuando especificas una constante, estás restringido a los tipos integrales integrados; los tipos bool, char y enum; y el tipo string.

Al final de cada cláusula case, debes especificar explícitamente hacia dónde debe dirigirse la ejecución, utilizando algún tipo de declaración de salto (a menos que tu código termine en un bucle infinito). Aquí están las opciones:

- break (salta al final de la declaración switch)

- goto case x (salta a otra cláusula case)
- goto default (salta a la cláusula default)
- Cualquier otra declaración de salto, como return, throw, continue o goto label

Cuando más de un valor debe ejecutar el mismo código, puedes listar los casos comunes de manera secuencial:

```
switch (cardNumber)
{
  case 13:
  case 12:
  case 11:
    Console.WriteLine ("Face card");
    break;
  default:
    Console.WriteLine ("Plain card");
    break;
}
```

Esta característica de la declaración switch puede ser fundamental para producir un código más limpio que múltiples declaraciones if-else.

Switching según los tipos



Cambiar según un tipo es un caso especial de cambiar según un patrón. Se han introducido varios otros patrones en las versiones recientes de C#; consulta "Patrones" en la página 222 para obtener una discusión completa al respecto.

También puedes cambiar según tipos (desde C# 7):

```
TellMeTheType (12);
TellMeTheType ("hello");
TellMeTheType (true);

void TellMeTheType (object x)   // object allows any type.
{
  switch (x)
  {
    case int i:
      Console.WriteLine ("It's an int!");
      Console.WriteLine ($"The square of {i} is {i * i}");
      break;
    case string s:
      Console.WriteLine ("It's a string");
      Console.WriteLine ($"The length of {s} is {s.Length}");
      break;
    default:
      Console.WriteLine ("I don't know what x is");
      break;
  }
}
```

(El tipo object permite una variable de cualquier tipo; se discute esto completamente en "Herencia" en la página 115 y "El tipo object" en la página 125).

Cada cláusula case especifica un tipo con el que hacer coincidir y una variable en la que asignar el valor tipado si la coincidencia tiene éxito (la "variable de patrón"). A diferencia de las constantes, no hay restricción en los tipos que puedes usar.

Puedes condicionar una cláusula case con la palabra clave when:

```
switch (x)
{
  case bool b when b == true:     // Fires only when b is true
    Console.WriteLine ("True!");
    break;
  case bool b:
    Console.WriteLine ("False!");
    break;
}
```

El orden de las cláusulas case puede importar cuando se cambia según el tipo (a diferencia de cuando se cambia según constantes). Este ejemplo daría un resultado diferente si invirtiéramos los dos casos (de hecho, ni siquiera se compilaría, porque el compilador determinaría que el segundo caso es inalcanzable). Una excepción a esta regla es la cláusula default, que siempre se ejecuta al final, independientemente de dónde aparezca.

Si deseas cambiar según un tipo, pero no te interesa su valor, puedes usar un descarte (_):

```
case DateTime _:
  Console.WriteLine ("It's a DateTime");
```

Puedes apilar múltiples cláusulas case. El Console.WriteLine en el siguiente código se ejecutará para cualquier tipo de punto flotante mayor que 1,000:

```
switch (x)
{
  case float f when f > 1000:
  case double d when d > 1000:
  case decimal m when m > 1000:
    Console.WriteLine ("We can refer to x here but not f or d or m");
    break;
}
```

En este ejemplo, el compilador nos permite utilizar las variables de patrón f, d y m solo en las cláusulas when. Cuando llamamos a Console.WriteLine, es desconocido cuál de esas tres variables será asignada, por lo que el compilador las pone fuera de alcance.

Puedes combinar constantes y patrones en la misma declaración switch. Y también puedes cambiar según el valor nulo:

```
case null:
  Console.WriteLine ("Nothing here");
  break;
```

Expresiones de cambio

A partir de C# 8, puedes usar el cambio en el contexto de una expresión. Suponiendo que cardNumber es de tipo int, lo siguiente ilustra su uso:

```
string cardName = cardNumber switch
{
  13 => "King",
  12 => "Queen",
  11 => "Jack",
  _ => "Pip card"   // equivalent to 'default'
};
```

Observa que la palabra clave switch aparece después del nombre de la variable y que las cláusulas case son expresiones (terminadas por comas) en lugar de declaraciones. Las expresiones de cambio son más compactas que sus contrapartes de declaración de cambio, y puedes usarlas en consultas LINQ (Capítulo 8).

Si omites la expresión predeterminada (_) y el cambio no coincide, se lanzará una excepción.

También puedes cambiar según múltiples valores (el patrón de tupla):

```
int cardNumber = 12;
string suite = "spades";

string cardName = (cardNumber, suite) switch
{
  (13, "spades") => "King of spades",
  (13, "clubs") => "King of clubs",
  ...
};
```

Hay muchas opciones adicionales posibles mediante el uso de patrones (ver "Patrones" en la página 222).

Declaraciones de iteración

C# permite que una secuencia de declaraciones se ejecute repetidamente con las declaraciones while, do-while, for y foreach.

Bucles while y do-while

Los bucles while ejecutan repetidamente un bloque de código mientras una expresión bool sea verdadera. La expresión se evalúa antes de ejecutar el bloque del bucle. Por ejemplo, lo siguiente escribe 012:

```
int i = 0;
while (i < 3)
{
  Console.Write (i);
  i++;
}
```

Los bucles do-while difieren de los bucles while solo en que evalúan la expresión después de ejecutar el bloque de instrucciones (asegurando que el bloque se ejecute al menos una vez). Aquí está el ejemplo anterior reescrito con un bucle do-while:

```
int i = 0;
do
{
  Console.WriteLine (i);
  i++;
}
while (i < 3);
```

Bucles for

Los bucles for son similares a los bucles while, pero tienen cláusulas especiales para la inicialización y la iteración de una variable de bucle. Un bucle for contiene tres cláusulas de la siguiente manera:

```
for (initialization-clause; condition-clause; iteration-clause)
  statement-or-statement-block
```

Aquí está qué hace cada cláusula:

Cláusula de inicialización:

Se ejecuta antes de que comience el bucle; se utiliza para inicializar una o más variables de iteración.

Cláusula de condición:

La expresión bool que, mientras sea verdadera, ejecutará el cuerpo del bucle.

Cláusula de iteración:

Se ejecuta después de cada iteración del bloque de instrucciones; se utiliza típicamente para actualizar la variable de iteración.

Por ejemplo, lo siguiente imprime los números del 0 al 2:

```
for (int i = 0; i < 3; i++)
  Console.WriteLine (i);
```

Lo siguiente imprime los primeros 10 números de Fibonacci (en los que cada número es la suma de los dos anteriores):

```
for (int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
{
  Console.WriteLine (prevFib);
  int newFib = prevFib + curFib;
  prevFib = curFib; curFib = newFib;
}
```

Cualquiera de las tres partes de la declaración for puede omitirse. Puedes implementar un bucle infinito como el siguiente (aunque se puede utilizar while(true) en su lugar):

```
for (;;)
  Console.WriteLine ("interrupt me");
```

Bucles foreach

Los bucles foreach iteran sobre cada elemento en un objeto enumerable. La mayoría de los tipos de .NET que representan un conjunto o lista de elementos son enumerables. Por ejemplo, tanto un arreglo como una cadena de texto son enumerables. Aquí tienes un ejemplo de iteración sobre los caracteres de una cadena, desde el primer carácter hasta el último:

```
foreach (char c in "beer")   // c is the iteration variable
  Console.WriteLine (c);
```

Aquí está la salida:

```
b
e
e
r
```

Definimos objetos enumerables en "Enumeración e Iteradores" en la página 189.

Instrucciones de salto

Las instrucciones de salto en C# son break, continue, goto, return y throw.



Las instrucciones de salto siguen las reglas de confiabilidad de las instrucciones try (ver "try Statements and Exceptions" en la página 181). Esto significa que:

- Un salto fuera de un bloque try siempre ejecuta el bloque finally del try antes de llegar al destino del salto.

- No se puede realizar un salto desde el interior hacia el exterior de un bloque finally (excepto a través de throw).

La declaración break

La declaración break finaliza la ejecución del cuerpo de una iteración o declaración switch:

```
int x = 0;
while (true)
{
  if (x++ > 5)
    break;      // break from the loop
}
// execution continues here after break
...
```

La declaración continue

La declaración continue omite las declaraciones restantes en un bucle y comienza de forma anticipada la siguiente iteración. El siguiente bucle omite los números pares:

```
for (int i = 0; i < 10; i++)
{
  if ((i % 2) == 0)       // If i is even,
    continue;             // continue with next iteration

  Console.Write (i + " ");
}

OUTPUT: 1 3 5 7 9
```

La declaración goto

La declaración goto trasfiere la ejecución a otra etiqueta dentro de un bloque de declaración. Su forma es la siguiente:

```
goto statement-label;
```

O cuando se utiliza dentro de una declaración switch:

```
goto case case-constant;    // (Only works with constants, not patterns)
```

Una etiqueta es un marcador en un bloque de código que precede a una declaración y se denota con un sufijo de dos puntos. El siguiente ejemplo itera los números del 1 al 5, imitando un bucle for:

```
int i = 1;
startLoop:
if (i <= 5)
{
  Console.Write (i + " ");
  i++;
  goto startLoop;
}

OUTPUT: 1 2 3 4 5
```

La declaración goto case case-constant trasfiere la ejecución a otro caso dentro de un bloque switch (ver "La declaración switch" en la página 79).

La declaración return

La declaración return sale del método y debe devolver una expresión del tipo de retorno del método si el método es de tipo no void:

```
decimal AsPercentage (decimal d)
{
  decimal p = d * 100m;
  return p;             // Return to the calling method with value
}
```

Una declaración return puede aparecer en cualquier lugar dentro de un método (excepto en un bloque finally) y puede ser utilizada más de una vez.

La declaración throw

La declaración throw lanza una excepción para indicar que se ha producido un error (ver "La declaración try y las excepciones" en la página 181):

```
if (w == null)
  throw new ArgumentNullException (...);
```

Declaraciones Misceláneas

La declaración using proporciona una sintaxis elegante para llamar al método Dispose en objetos que implementan IDisposable, dentro de un bloque finally (ver "La declaración try y las excepciones" en la página 181 y "IDisposable, Dispose y Close" en la página 557).



C# sobrecarga la palabra clave "using" para tener significados independientes en diferentes contextos. Específicamente, la directiva "using" es diferente de la declaración "using".

La declaración "lock" es un atajo para llamar a los métodos Enter y Exit de la clase Monitor (consultar los capítulos 14 y 23).

Espacios de nombres

Un espacio de nombres es un dominio para los nombres de tipos. Los tipos suelen organizarse en espacios de nombres jerárquicos, lo que facilita su búsqueda y evita conflictos. Por ejemplo, el tipo RSA que maneja la encriptación de clave pública se define dentro del siguiente espacio de nombres:

```
System.Security.Cryptography
```

Un espacio de nombres forma una parte integral del nombre de un tipo. El siguiente código llama al método Create de RSA:

```
System.Security.Cryptography.RSA rsa =
  System.Security.Cryptography.RSA.Create();
```



Los espacios de nombres son independientes de los ensamblados, que son archivos .dll que sirven como unidades de implementación (descritos en el Capítulo 17). Los espacios de nombres tampoco tienen impacto en la visibilidad de los miembros, como público, interno, privado, y así sucesivamente.

La palabra clave "namespace" define un espacio de nombres para los tipos dentro de ese bloque; por ejemplo:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
  class Class2 {}
}
```

Los puntos en el espacio de nombres indican una jerarquía de espacios de nombres anidados. El siguiente código es semánticamente idéntico al ejemplo anterior:

```
namespace Outer
{
  namespace Middle
  {
    namespace Inner
    {
      class Class1 {}
      class Class2 {}
    }
  }
}
```

Puedes referirte a un tipo con su nombre completamente calificado, que incluye todos los espacios de nombres desde el más externo hasta el más interno. Por ejemplo, podríamos referirnos a Class1 en el ejemplo anterior como Outer.Middle.Inner.Class1.

Los tipos que no están definidos en ningún espacio de nombres se dice que residen en el espacio de nombres global. El espacio de nombres global también incluye los espacios de nombres de nivel superior, como Outer en nuestro ejemplo.

Los espacios de nombres en el ámbito de archivos (C# 10)

A menudo, querrás que todos los tipos en un archivo estén definidos en un solo espacio de nombres:

```
namespace MyNamespace
{
  class Class1 {}
  class Class2 {}
}
```

A partir de C# 10, puedes lograr esto con un espacio de nombres en el ámbito de archivos:

```
namespace MyNamespace;  // Applies to everything that follows in the file.

class Class1 {}         // inside MyNamespace
class Class2 {}         // inside MyNamespace
```

Los espacios de nombres en el ámbito de archivos reducen el desorden y eliminan un nivel de sangría innecesario.

La directiva using

La directiva using importa un espacio de nombres, lo que te permite referirte a los tipos sin utilizar sus nombres completamente calificados. El siguiente ejemplo importa el espacio de nombres Outer.Middle.Inner del ejemplo anterior:

```
using Outer.Middle.Inner;

Class1 c;    // Don't need fully qualified name
```



Es legal (y a menudo deseable) definir el mismo nombre de tipo en diferentes espacios de nombres. Sin embargo, normalmente solo lo harías si es poco probable que un consumidor quiera importar ambos espacios de nombres al mismo tiempo. Un buen ejemplo es la clase TextBox, que está definida tanto en System.Windows.Controls (WPF) como en System.Windows.Forms (Windows Forms).

Una directiva using puede estar anidada dentro de un espacio de nombres para limitar el alcance de la directiva.

La directiva global using (C# 10)

A partir de C# 10, si prefijas una directiva using con la palabra clave global, la directiva se aplicará a todos los archivos en el proyecto o en la unidad de compilación:

```
global using System;
global using System.Collection.Generic;
```

Esto te permite centralizar las importaciones comunes y evitar repetir las mismas directivas en cada archivo.

Las directivas globales using deben preceder a las directivas no globales y no pueden aparecer dentro de declaraciones de espacios de nombres. La directiva global se puede utilizar con using static.

Implicit global usings

A partir de .NET 6, los archivos de proyecto permiten directivas using globales implícitas. Si se establece el elemento ImplicitUsings en true en el archivo de proyecto (lo cual es el valor predeterminado para los nuevos proyectos), se importarán automáticamente los siguientes espacios de nombres:

```
System
System.Collections.Generic
System.IO
System.Linq
System.Net.Http
System.Threading
System.Threading.Tasks
```

Se importarán espacios de nombres adicionales según el SDK del proyecto (Web, Windows Forms, WPF, etc.).

using static

La directiva using static importa un tipo en lugar de un espacio de nombres. Todos los miembros estáticos del tipo importado pueden usarse sin calificación. En el siguiente ejemplo, llamamos al método WriteLine estático de la clase Console sin necesidad de hacer referencia al tipo:

```
using static System.Console;

WriteLine ("Hello");
```

La directiva using static importa todos los miembros estáticos accesibles del tipo, incluyendo campos, propiedades y tipos anidados. También se puede aplicar esta directiva a tipos de enumeración, en cuyo caso se importan sus miembros. Por ejemplo, si importamos el siguiente tipo de enumeración:

```
using static System.Windows.Visibility;
```

Podemos especificar Hidden en lugar de Visibility.Hidden:

```
var textBox = new TextBox { Visibility = Hidden };   // XAML-style
```

En caso de que surja una ambigüedad entre múltiples importaciones estáticas, el compilador de C# no es lo suficientemente inteligente como para inferir el tipo correcto a partir del contexto y generará un error.

Reglas dentro de un espacio de nombres

Alcance de nombres

Los nombres declarados en espacios de nombres externos pueden usarse sin calificación dentro de espacios de nombres internos. En este ejemplo, Class1 no necesita calificación dentro de Inner:

```
namespace Outer
{
  class Class1 {}

  namespace Inner

  {
    class Class2 : Class1  {}
  }
}
```

Si deseas hacer referencia a un tipo en una rama diferente de la jerarquía de tu espacio de nombres, puedes usar un nombre parcialmente calificado. En el siguiente ejemplo, basamos SalesReport en Common.ReportBase:

```
namespace MyTradingCompany
{
  namespace Common
  {
    class ReportBase {}
  }
  namespace ManagementReporting
  {
    class SalesReport : Common.ReportBase  {}
  }
}
```

Ocultamiento de nombres

Si el mismo nombre de tipo aparece tanto en un espacio de nombres interno como en uno externo, el nombre interno tiene prioridad. Para hacer referencia al tipo en el espacio de nombres externo, debes calificar su nombre:

```
namespace Outer
{
  class Foo { }

  namespace Inner
  {
    class Foo { }

    class Test
    {
      Foo f1;          // = Outer.Inner.Foo
      Outer.Foo f2;    // = Outer.Foo
    }
  }
}
```



Todos los nombres de tipo se convierten a nombres completamente calificados en tiempo de compilación. El código de Lenguaje de Intermedio (IL) no contiene nombres sin calificar o parcialmente calificados.

Puedes repetir una declaración de espacio de nombres siempre y cuando los nombres de tipo dentro de los espacios de nombres no entren en conflicto:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}


 namespace Outer.Middle.Inner
 {
   class Class2 {}
 }
```

Incluso podemos dividir el ejemplo en dos archivos fuente de manera que podamos compilar cada clase en un ensamblado diferente.

Archivo fuente 1:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}
```

Archivo fuente 2:

```
namespace Outer.Middle.Inner
{
  class Class2 {}
}
```

Directivas using anidadas

Puedes anidar una directiva using dentro de un espacio de nombres. Esto te permite delimitar el alcance de la directiva using dentro de una declaración de espacio de nombres. En el siguiente ejemplo, Class1 es visible en un ámbito pero no en otro:

```
namespace N1
{
  class Class1 {}
}

namespace N2
{
  using N1;

  class Class2 : Class1 {}
}

namespace N2
{
  class Class3 : Class1 {}   // Compile-time error
}
```

Alias de tipos y espacios de nombres

Importar un espacio de nombres puede resultar en colisión de nombres de tipo. En lugar de importar todo el espacio de nombres, puedes importar solo los tipos específicos que necesitas, asignando a cada tipo un alias:

```
using PropertyInfo2 = System.Reflection.PropertyInfo;
class Program { PropertyInfo2 p; }
```

Un espacio de nombres completo puede recibir un alias de la siguiente manera:

```
using R = System.Reflection;
class Program { R.PropertyInfo p; }
```

Funciones avanzadas de espacios de nombres

Extern

Los alias externos permiten que tu programa haga referencia a dos tipos con el mismo nombre completamente calificado (es decir, el espacio de nombres y el nombre del tipo son idénticos). Este es un escenario inusual y solo puede ocurrir cuando los dos tipos provienen de ensamblados diferentes.

Considera el siguiente ejemplo.

Biblioteca 1, compilada en Widgets1.dll:

```
namespace Widgets
{
  public class Widget {}
}
```

Biblioteca 2, compilada en Widgets2.dll:

```
namespace Widgets
{
  public class Widget {}
}
```

Aplicación, que hace referencia a Widgets1.dll y Widgets2.dll:

```
using Widgets;

Widget w = new Widget();
```

La aplicación no puede compilar porque Widget es ambiguo. Los alias externos pueden resolver la ambigüedad. El primer paso es modificar el archivo .csproj de la aplicación, asignando un alias único a cada referencia:

```
<ItemGroup>
  <Reference Include="Widgets1">
    <Aliases>W1</Aliases>
  </Reference>
  <Reference Include="Widgets2">
    <Aliases>W2</Aliases>
  </Reference>
</ItemGroup>
```

El segundo paso es utilizar la directiva extern alias:

```
extern alias W1;
extern alias W2;

W1.Widgets.Widget w1 = new W1.Widgets.Widget();
W2.Widgets.Widget w2 = new W2.Widgets.Widget();
```

Calificadores de alias de espacio de nombres

Como mencionamos anteriormente, los nombres en espacios de nombres internos ocultan los nombres en espacios de nombres externos. Sin embargo, a veces incluso el uso de un nombre de tipo completamente calificado no resuelve el conflicto. Considera el siguiente ejemplo:

```
namespace N
{
  class A
  {
    static void Main() => new A.B();     // Instantiate class B
    public class B {}                    // Nested type
  }
}

namespace A
{
  class B {}
}
```

El método Main podría estar instanciando tanto la clase B anidada como la clase B dentro del espacio de nombres A. El compilador siempre da mayor prioridad a los identificadores en el espacio de nombres actual, en este caso la clase B anidada.

Para resolver tales conflictos, se puede calificar un nombre de espacio de nombres, en relación a uno de los siguientes:

- El espacio de nombres global: la raíz de todos los espacios de nombres (identificado con la palabra clave contextual global).
- El conjunto de alias externos.

El token "::" realiza la calificación del alias de espacio de nombres. En este ejemplo, calificamos utilizando el espacio de nombres global (esto se ve con mayor frecuencia en código generado automáticamente para evitar conflictos de nombres):

```
namespace N
{
  class A
  {
    static void Main()
    {
      System.Console.WriteLine (new A.B());
      System.Console.WriteLine (new global::A.B());
    }

    public class B {}
  }
}

namespace A
{
  class B {}
}
```

Aquí tienes un ejemplo de calificación con un alias (adaptado del ejemplo en "Extern"):

```
extern alias W1;
extern alias W2;

W1::Widgets.Widget w1 = new W1::Widgets.Widget();
W2::Widgets.Widget w2 = new W2::Widgets.Widget();
```

3 Creación de tipos en C#

En este capítulo, nos adentramos en los tipos y miembros de tipo.

Clases

Una clase es el tipo de referencia más común. La declaración más simple de una clase es la siguiente:

```
class YourClassName
{
}
```

Una clase más compleja opcionalmente tiene lo siguiente:

Preceding the keyword `class`	*Attributes* and *class modifiers*. The non-nested class modifiers are `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe`, and `partial`.
Following `YourClassName`	*Generic type parameters* and *constraints*, a *base class*, and *interfaces*.
Within the braces	*Class members* (these are *methods*, *properties*, *indexers*, *events*, *fields*, *constructors*, *overloaded operators*, *nested types*, and a *finalizer*).

Este capítulo cubre todos estos constructos excepto los atributos, funciones de operador y la palabra clave unsafe, que se tratan en el Capítulo 4. Las siguientes secciones enumeran cada uno de los miembros de la clase.

Campos

Un campo es una variable que es miembro de una clase o estructura; por ejemplo:

```
class Octopus
{
  string name;
  public int Age = 10;
}
```

Los campos permiten los siguientes modificadores:

Static modifier	`static`
Access modifiers	`public internal private protected`
Inheritance modifier	`new`
Unsafe code modifier	`unsafe`
Read-only modifier	`readonly`
Threading modifier	`volatile`

Existen dos convenciones populares de nombres para campos privados: en camel case (por ejemplo, firstName) y en camel case con un guión bajo (_firstName). Esta última convención te permite distinguir rápidamente los campos privados de los parámetros y variables locales.

El modificador readonly

El modificador readonly evita que un campo sea modificado después de la construcción. Un campo de solo lectura solo puede ser asignado en su declaración o dentro del constructor de la clase que lo contiene.

Inicialización de campos

La inicialización de campos es opcional. Un campo no inicializado tiene un valor predeterminado (0, '\0', null, false). Los inicializadores de campo se ejecutan antes de los constructores:

```
public int Age = 10;
```

Un inicializador de campo puede contener expresiones y llamar a métodos:

```
static readonly string TempFolder = System.IO.Path.GetTempPath();
```

Declarar múltiples campos juntos

Por conveniencia, puedes declarar múltiples campos del mismo tipo en una lista separada por comas. Esta es una forma conveniente de compartir los mismos modificadores y atributos para todos los campos:

```
static readonly int legs = 8,
                    eyes = 2;
```

Constantes

Una constante se evalúa estáticamente en tiempo de compilación y el compilador sustituye literalmente su valor siempre que se use (similar a una macro en C++). Una constante puede ser de tipo bool, char, string, cualquiera de los tipos numéricos incorporados o un tipo enum.

Una constante se declara con la palabra clave const y debe ser inicializada con un valor. Por ejemplo:

```
public class Test
{
  public const string Message = "Hello World";
}
```

Una constante puede desempeñar un papel similar al de un campo estático de solo lectura (static readonly), pero es mucho más restrictiva, tanto en los tipos que puedes usar como en la semántica de inicialización del campo. Además, una constante difiere de un campo estático de solo lectura en que la evaluación de la constante ocurre en tiempo de compilación. Por lo tanto, el código que mencionas:

```
public static double Circumference (double radius)
{
  return 2 * System.Math.PI * radius;
}
```

se compila a:

```
public static double Circumference (double radius)
{
  return 6.2831853071795862 * radius;
}
```

Tiene sentido que PI sea una constante porque su valor está predefinido en tiempo de compilación. En contraste, el valor de un campo estático de solo lectura puede diferir cada vez que se ejecuta el programa:

```
static readonly DateTime StartupTime = DateTime.Now;
```



Un campo estático de solo lectura (static readonly) también es ventajoso cuando se expone a otras bibliotecas un valor que podría cambiar en una versión posterior. Por ejemplo, supongamos que la biblioteca X expone una constante de la siguiente manera:

```
public const decimal ProgramVersion = 2.3;
```

Si la biblioteca Y hace referencia a X y utiliza esta constante, el valor 2.3 se incorporará en la biblioteca Y cuando se compile. Esto significa que si X se vuelve a compilar con la constante establecida en 2.4, Y seguirá utilizando el valor antiguo de 2.3 hasta que se vuelva a compilar Y. Un campo estático de solo lectura evita este problema.

Otra forma de ver esto es que cualquier valor que pueda cambiar en el futuro no es constante por definición; por lo tanto, no debería representarse como tal.

Las constantes también se pueden declarar localmente dentro de un método:

```
void Test()
{
  const double twoPI = 2 * System.Math.PI;
  ...
}
```

Las constantes no locales permiten los siguientes modificadores:

Access modifiers	public internal private protected
Inheritance modifier	new

Metodos

Un método realiza una acción en una serie de declaraciones. Un método puede recibir datos de entrada del llamador al especificar parámetros, y devolver datos al llamador al especificar un tipo de retorno. Un método puede especificar un tipo de retorno void, indicando que no devuelve ningún valor al llamador. Un método también puede devolver datos al llamador mediante parámetros ref/out.

La firma de un método debe ser única dentro del tipo. La firma de un método comprende su nombre y los tipos de parámetros en orden (pero no los nombres de los parámetros ni el tipo de retorno).

Los métodos permiten los siguientes modificadores:

Static modifier	`static`
Access modifiers	`public internal private protected`
Inheritance modifiers	`new virtual abstract override sealed`
Partial method modifier	`partial`
Unmanaged code modifiers	`unsafe extern`
Asynchronous code modifier	`async`

Métodos de expresión corporal

Un método que consiste en una sola expresión, como:

```
int Foo (int x) { return x * 2; }
```

puede escribirse de manera más concisa como un método de expresión corporal. Una flecha gorda reemplaza las llaves y la palabra clave return:

```
int Foo (int x) => x * 2;
```

Los métodos de expresión corporal también pueden tener un tipo de retorno void:

```
void Foo (int x) => Console.WriteLine (x);
```

Métodos locales

Puedes definir un método dentro de otro método:

```
void WriteCubes()
{
  Console.WriteLine (Cube (3));
  Console.WriteLine (Cube (4));
  Console.WriteLine (Cube (5));


  int Cube (int value) => value * value * value;
}
```

El método local (Cube, en este caso) solo es visible para el método que lo contiene (WriteCubes). Esto simplifica el tipo que lo contiene e indica de inmediato a cualquier persona que observe el código que Cube no se utiliza en ningún otro lugar. Otro beneficio de los métodos locales es que pueden acceder a las variables locales y parámetros del método que los contiene. Esto tiene varias consecuencias, que se describen en detalle en "Capturing Outer Variables" en la página 176.

Los métodos locales pueden aparecer dentro de otros tipos de funciones, como los accesores de propiedades, constructores, etc. Incluso puedes poner métodos locales dentro de otros métodos locales y dentro de expresiones lambda que utilicen un bloque de sentencias (Capítulo 4). Los métodos locales pueden ser iteradores (Capítulo 4) o asíncronos (Capítulo 14).

Métodos locales estáticos

Agregar el modificador static a un método local (a partir de C# 8) evita que acceda a las variables locales y parámetros del método que lo contiene. Esto ayuda a reducir el acoplamiento y evita que el método local haga referencia accidentalmente a variables del método contenedor.

Métodos locales y declaraciones en el nivel superior

Cualquier método que declares en declaraciones en el nivel superior se trata como un método local. Esto significa que (a menos que se marque como static) puede acceder a las variables en las declaraciones en el nivel superior:

```
int x = 3;
Foo();

void Foo() => Console.WriteLine (x);
```

Sobrecarga de métodos



Los métodos locales no pueden ser sobrecargados. Esto significa que los métodos declarados en declaraciones de nivel superior (que se tratan como métodos locales) no pueden ser sobrecargados.

Un tipo puede sobrecargar métodos (definir múltiples métodos con el mismo nombre) siempre y cuando las firmas sean diferentes. Por ejemplo, los siguientes métodos pueden coexistir en el mismo tipo:

```
void Foo (int x) {...}
void Foo (double x) {...}
void Foo (int x, float y) {...}
void Foo (float x, int y) {...}
```

Sin embargo, las siguientes parejas de métodos no pueden coexistir en el mismo tipo, porque el tipo de retorno y el modificador "params" no forman parte de la firma de un método:

```
void  Foo (int x) {...}
float Foo (int x) {...}          // Compile-time error

void  Goo (int[] x) {...}
void  Goo (params int[] x) {...}  // Compile-time error
```

Si un parámetro se pasa por valor o por referencia también forma parte de la firma. Por ejemplo, Foo(int) puede coexistir con Foo(ref int) o Foo(out int). Sin embargo, Foo(ref int) y Foo(out int) no pueden coexistir:

```
void Foo (int x) {...}
void Foo (ref int x) {...}     // OK so far
void Foo (out int x) {...}     // Compile-time error
```

Constructores de instancia

Los constructores ejecutan código de inicialización en una clase o estructura. Un constructor se define como un método, excepto que el nombre del método y el tipo de retorno se reducen al nombre de la clase o estructura contenedora:

```
Panda p = new Panda ("Petey");   // Call constructor

public class Panda
{
  string name;                   // Define field
  public Panda (string n)        // Define constructor
  {
    name = n;                    // Initialization code (set up field)
  }
}
```

Los constructores de instancia permiten los siguientes modificadores:

Access modifiers	`public internal private protected`
Unmanaged code modifiers	`unsafe extern`

Los constructores de una sola declaración también se pueden escribir como miembros con cuerpo de expresión:

```
public Panda (string n) => name = n;
```

Sobrecarga de constructores

Una clase o estructura puede tener constructores sobrecargados. Para evitar duplicación de código, un constructor puede llamar a otro usando la palabra clave "this":

```
using System;

public class Wine
{
  public decimal Price;
  public int Year;
  public Wine (decimal price) { Price = price; }


  public Wine (decimal price, int year) : this (price) { Year = year; }
}
```

Cuando un constructor llama a otro, el constructor llamado se ejecuta primero.

Puedes pasar una expresión a otro constructor de la siguiente manera:

```
public Wine (decimal price, DateTime year) : this (price, year.Year) { }
```

La expresión en sí no puede hacer uso de la referencia "this", por ejemplo, para llamar a un método de instancia. (Esto se cumple porque el objeto no ha sido inicializado por el constructor en esta etapa, por lo que cualquier método al que llames es probable que falle). Sin embargo, puede llamar a métodos estáticos.

Constructores implícitos sin parámetros

Para las clases, el compilador de C# genera automáticamente un constructor público sin parámetros si y solo si no defines ningún constructor. Sin embargo, tan pronto como definas al menos un constructor, el constructor sin parámetros ya no se genera automáticamente.

Orden de inicialización de constructores y campos

Anteriormente vimos que los campos pueden inicializarse con valores predeterminados en su declaración:

```
class Player
{
  int shields = 50;   // Initialized first
  int health = 100;   // Initialized second
}
```

Las inicializaciones de campos ocurren antes de que se ejecute el constructor y en el orden de declaración de los campos.

Constructores no públicos

Los constructores no necesitan ser públicos. Una razón común para tener un constructor no público es controlar la creación de instancias mediante una llamada a un método estático. El método estático podría utilizarse para devolver un objeto de un grupo en lugar de crear un objeto nuevo, o devolver varias subclases basadas en argumentos de entrada:

```
public class Class1
{
  Class1() {}                          // Private constructor
  public static Class1 Create (...)
  {
    // Perform custom logic here to return an instance of Class1
    ...
  }
}
```

Destructor(Deconstructors)

Un destructor (también llamado método destructor) actúa como un opuesto aproximado a un constructor: mientras que un constructor generalmente toma un conjunto de valores (como parámetros) y los asigna a los campos, un destructor hace lo contrario y asigna los campos de vuelta a un conjunto de variables.

Un método de deconstrucción debe llamarse Deconstruct y tener uno o más parámetros de salida (out), como en la siguiente clase:

```
class Rectangle
{
  public readonly float Width, Height;

  public Rectangle (float width, float height)
  {
    Width = width;
    Height = height;
  }

  public void Deconstruct (out float width, out float height)
  {
    width = Width;
    height = Height;
  }
}
```

La siguiente sintaxis especial llama al destructor:

```
var rect = new Rectangle (3, 4);
(float width, float height) = rect;          // Deconstruction
Console.WriteLine (width + " " + height);    // 3 4
```

La segunda línea es la llamada de deconstrucción. Crea dos variables locales y luego llama al método Deconstruct. Nuestra llamada de deconstrucción es equivalente a lo siguiente:

```
float width, height;
rect.Deconstruct (out width, out height);
```

O también:

```
rect.Deconstruct (out var width, out var height);
```

Las llamadas de deconstrucción permiten la inferencia de tipos implícita, por lo que podríamos acortar nuestra llamada a esto:

```
(var width, var height) = rect;
```

O simplemente esto:

```
var (width, height) = rect;
```

 Puedes utilizar el símbolo de descarte (_) de C# si no estás interesado en una o más variables:

```
var (_, height) = rect;
```

Esto indica mejor tu intención que declarar una variable que nunca usarás.

Si las variables en las que estás desestructurando ya están definidas, puedes omitir por completo los tipos:

```
float width, height;
(width, height) = rect;
```

Esto se llama una asignación de desestructuración. Puedes utilizar una asignación de desestructuración para simplificar el constructor de tu clase:

```
public Rectangle (float width, float height) =>
  (Width, Height) = (width, height);
```

Puedes ofrecer al llamador una variedad de opciones de desestructuración mediante la sobrecarga del método Deconstruct.



El método Deconstruct puede ser un método de extensión (ver "Extension Methods" en la página 202). Este es un truco útil si deseas desestructurar tipos que no has creado tú mismo.

Desde C# 10, puedes combinar variables existentes y nuevas al desestructurar:

```
double x1 = 0;
(x1, double y2) = rect;
```

Inicializadores de objetos

Para simplificar la inicialización de objetos, cualquier campo o propiedad accesible de un objeto puede establecerse mediante un inicializador de objetos justo después de la construcción. Por ejemplo, considera la siguiente clase:

```
public class Bunny
{
  public string Name;
  public bool LikesCarrots;
  public bool LikesHumans;

  public Bunny () {}
  public Bunny (string n) { Name = n; }
}
```

Usando inicializadores de objetos, puedes instanciar objetos Bunny de la siguiente manera:

```
// Note parameterless constructors can omit empty parentheses
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false };
Bunny b2 = new Bunny ("Bo")     { LikesCarrots=true, LikesHumans=false };
```

El código para construir b1 y b2 es precisamente equivalente a lo siguiente:

```
Bunny temp1 = new Bunny();    // temp1 is a compiler-generated name
temp1.Name = "Bo";
temp1.LikesCarrots = true;
temp1.LikesHumans = false;
Bunny b1 = temp1;

Bunny temp2 = new Bunny ("Bo");
temp2.LikesCarrots = true;
temp2.LikesHumans = false;
Bunny b2 = temp2;
```

Las variables temporales se utilizan para asegurar que si se produce una excepción durante la inicialización, no termines con un objeto medio inicializado.

Inicializadores de objetos versus Parámetros opcionales

En lugar de usar inicializadores de objetos, podríamos hacer que el constructor de Bunny acepte parámetros opcionales:

```
public Bunny (string name,
              bool likesCarrots = false,
              bool likesHumans = false)
{
  Name = name;
  LikesCarrots = likesCarrots;
  LikesHumans = likesHumans;
}
```

Esto nos permitiría construir un objeto Bunny de la siguiente manera:

```
Bunny b1 = new Bunny (name: "Bo",
                      likesCarrots: true);
```

Históricamente, este enfoque podría ser ventajoso ya que nos permitía hacer que los campos (o propiedades, que explicaremos en breve) de Bunny fueran de solo lectura. Hacer que los campos o propiedades sean de solo lectura es una buena práctica cuando no hay una razón válida para que cambien a lo largo de la vida del objeto. Sin embargo, como veremos pronto en nuestra discusión sobre propiedades, el modificador init que se introdujo en C# 9 nos permite lograr este objetivo con inicializadores de objetos.

Los parámetros opcionales tienen dos inconvenientes. El primero es que su uso en constructores permite tipos de solo lectura, pero no permite una mutación no destructiva de manera fácil. (Cubriremos la mutación no destructiva, y la solución a este problema, en "Registros" en la página 211).

El segundo inconveniente de los parámetros opcionales es que, cuando se utilizan en bibliotecas públicas, dificultan la compatibilidad hacia atrás. Esto se debe a que agregar un parámetro opcional en una fecha posterior rompe la compatibilidad binaria de la biblioteca con los consumidores existentes. (Esto es especialmente importante cuando una biblioteca se publica

en NuGet: el problema se vuelve intratable cuando un consumidor hace referencia a los paquetes A y B, si A y B dependen de versiones incompatibles de L).

El problema radica en que cada valor de parámetro opcional se establece en el sitio de llamada. En otras palabras, C# traduce nuestra llamada al constructor de la siguiente manera:

```
Bunny b1 = new Bunny ("Bo", true, false);
```

Esto es problemático si instanciamos la clase Bunny desde otra biblioteca y luego modificamos Bunny agregando otro parámetro opcional, como likesCats. A menos que la biblioteca de referencia también sea recompilada, continuará llamando al constructor (ahora inexistente) con tres parámetros y fallará en tiempo de ejecución. (Un problema más sutil es que si cambiamos el valor de uno de los parámetros opcionales, los llamadores en otras bibliotecas seguirán utilizando el valor opcional anterior hasta que sean recompilados).

La referencia "this"

La referencia "this" se refiere a la instancia misma. En el siguiente ejemplo, el método "Marry" utiliza "this" para establecer el campo "mate" de la pareja:

```
public class Panda
{
  public Panda Mate;

  public void Marry (Panda partner)
  {
    Mate = partner;
    partner.Mate = this;
  }
}
```

La referencia "this" también desambigua una variable local o un parámetro de un campo. Por ejemplo:

```
public class Test
{
  string name;
  public Test (string name) { this.name = name; }
}
```

La referencia "this" es válida únicamente dentro de miembros no estáticos de una clase o estructura.

Propiedades

Las propiedades se ven como campos desde el exterior, pero internamente contienen lógica, similar a los métodos. Por ejemplo, no puedes determinar al observar el siguiente código si "CurrentPrice" es un campo o una propiedad:

```
Stock msft = new Stock();
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

Una propiedad se declara como un campo, pero con un bloque "get/set" agregado. Así es cómo se implementaría "CurrentPrice" como una propiedad:

```
public class Stock
{
  decimal currentPrice;           // The private "backing" field

  public decimal CurrentPrice     // The public property
  {
    get { return currentPrice; }
    set { currentPrice = value; }
  }
}
```

Los identificadores "get" y "set" denotan los accesores de una propiedad. El accesor "get" se ejecuta cuando se lee la propiedad. Debe devolver un valor del tipo de la propiedad. El accesor "set" se ejecuta cuando se asigna un valor a la propiedad. Tiene un parámetro implícito llamado "value" del tipo de la propiedad, que generalmente se asigna a un campo privado (en este caso, "currentPrice").

Aunque las propiedades se acceden de la misma manera que los campos, difieren en que brindan al implementador un control completo sobre la obtención y configuración de su valor. Este control permite al implementador elegir la representación interna necesaria sin exponer los detalles internos al usuario de la propiedad. En este ejemplo, el método "set" podría lanzar una excepción si el valor se encuentra fuera de un rango válido.



A lo largo de este libro, utilizamos campos públicos de manera extensiva para mantener los ejemplos libres de distracciones. Sin embargo, en una aplicación real, generalmente es preferible utilizar propiedades públicas en lugar de campos públicos para promover la encapsulación.

Las propiedades permiten los siguientes modificadores:

Static modifier	`static`
Access modifiers	`public internal private protected`
Inheritance modifiers	`new virtual abstract override sealed`
Unmanaged code modifiers	`unsafe extern`

Propiedades de solo lectura y calculadas

Una propiedad es de solo lectura si especifica únicamente un accesor "get", y es de solo escritura si especifica únicamente un accesor "set". Las propiedades de solo escritura se utilizan raramente.

Normalmente, una propiedad tiene un campo de respaldo dedicado para almacenar los datos subyacentes.

Sin embargo, también es posible calcular el valor de una propiedad a partir de otros datos:

```
decimal currentPrice, sharesOwned;

public decimal Worth
{
  get { return currentPrice * sharesOwned; }
}
```

Propiedades expresadas en una sola línea

Puedes declarar una propiedad de solo lectura, como la del ejemplo anterior, de manera más concisa utilizando una propiedad expresada en una sola línea. Una flecha gruesa reemplaza todas las llaves, así como las palabras clave "get" y "return":

```
public decimal Worth => currentPrice * sharesOwned;
```

Con un poco de sintaxis adicional, los accesores de escritura también pueden ser expresados en una sola línea:

```
public decimal Worth
{
  get => currentPrice * sharesOwned;
  set => sharesOwned = value / currentPrice;
}
```

Propiedades automáticas

La implementación más común para una propiedad es tener un getter y/o un setter que simplemente lee y escribe en un campo privado del mismo tipo que la propiedad. Una declaración de propiedad automática instruye al compilador a proporcionar esta implementación. Podemos mejorar el primer ejemplo de esta sección declarando "CurrentPrice" como una propiedad automática:

```
public class Stock
{
  ...
  public decimal CurrentPrice { get; set; }
}
```

El compilador genera automáticamente un campo privado de respaldo con un nombre generado por el compilador al que no se puede hacer referencia. El accesor de escritura puede ser marcado como privado o protegido si deseas exponer la propiedad como de solo lectura para otros tipos.

Las propiedades automáticas se introdujeron en C# 3.0.

Inicializadores de propiedades

Puedes agregar un inicializador de propiedades a las propiedades automáticas, al igual que con los campos:

```
public decimal CurrentPrice { get; set; } = 123;
```

Esto asigna un valor inicial de 123 a "CurrentPrice". Las propiedades con un inicializador pueden ser de solo lectura:

```
public int Maximum { get; } = 999;
```

Al igual que con los campos de solo lectura, las propiedades automáticas de solo lectura también se pueden asignar en el constructor del tipo. Esto es útil para crear tipos inmutables (de solo lectura).

Accesibilidad de "get" y "set"

Los accesores "get" y "set" pueden tener diferentes niveles de acceso. El caso de uso típico es tener una propiedad pública con un modificador de acceso interno o privado en el setter:

```
public class Foo
{
  private decimal x;
  public decimal X
  {
    get         { return x;  }
    private set { x = Math.Round (value, 2); }
  }
}
```

Setters de solo inicializacion

A partir de C# 9, puedes declarar un accesor de propiedad con "init" en lugar de "set", lo que crea propiedades de solo inicialización:

```
public class Note
{
  public int Pitch    { get; init; } = 20;   // “Init-only” property
  public int Duration { get; init; } = 100;  // “Init-only” property
}
```

Estas propiedades de solo inicialización actúan como propiedades de solo lectura, excepto que también se pueden establecer mediante un inicializador de objetos:

```
var note = new Note { Pitch = 50 };
```

Después de eso, la propiedad no se puede modificar:

```
note.Pitch = 200;  // Error - init-only setter!
```

Las propiedades de solo inicialización no se pueden establecer incluso desde dentro de su clase, excepto a través de su inicializador de propiedad, el constructor u otro accesor de solo inicialización.

La alternativa a las propiedades de solo inicialización es tener propiedades de solo lectura que se inicializan mediante un constructor:

```
public class Note
{
  public int Pitch    { get; }
  public int Duration { get; }

  public Note (int pitch = 20, int duration = 100)
  {
    Pitch = pitch; Duration = duration;
  }
}
```

Si la clase forma parte de una biblioteca pública, este enfoque dificulta la versión, ya que agregar un parámetro opcional al constructor en una fecha posterior rompe la compatibilidad binaria con los consumidores (mientras que agregar una nueva propiedad de solo inicialización no rompe nada).



Las propiedades de solo inicialización tienen otra ventaja significativa, que es permitir la mutación no destructiva cuando se utilizan en conjunto con los registros (ver "Registros" en la página 211).

Al igual que con los accesores de escritura ordinarios, los accesores de solo inicialización pueden proporcionar una implementación:

```
public class Note
{
  readonly int _pitch;
  public int Pitch { get => _pitch; init => _pitch = value; }
  ...
```

Observa que el campo `_tono` es de solo lectura: los accesores de solo inicialización tienen permitido modificar campos de solo lectura en su propia clase. (Sin esta característica, `_tono` tendría que ser escribible y la clase no sería internamente inmutable).



Cambiar el accesor de una propiedad de "init" a "set" (o viceversa) es un cambio que rompe la compatibilidad binaria: cualquier persona que haga referencia a tu ensamblado deberá volver a compilar su ensamblado.

Esto no debería ser un problema al crear tipos completamente inmutables, ya que tu tipo nunca requerirá propiedades con un accesor de escritura (editable).

Propiedad de implementación CLR

La implementación interna de los accesores de propiedades en C# se compila a métodos llamados get_XXX y set_XXX:

```
public decimal get_CurrentPrice {...}
public void set_CurrentPrice (decimal value) {...}
```

Un accesor "init" se procesa como un accesor "set", pero con una bandera adicional codificada en los metadatos de "modreq" del accesor "set" (ver "Propiedades de solo inicialización" en la página 794).

Los accesores de propiedades simples y no virtuales son inlinedos por el compilador Just-in-Time (JIT), eliminando cualquier diferencia de rendimiento entre acceder a una propiedad y un campo. La inlining es una optimización en la que una llamada a método se reemplaza por el cuerpo de ese método.

Indexadores

Los indexadores proporcionan una sintaxis natural para acceder a elementos en una clase o estructura que encapsula una lista o diccionario de valores. Los indexadores son similares a las propiedades, pero se acceden a través de un argumento de índice en lugar de un nombre de propiedad. La clase `string` tiene un indexador que te permite acceder a cada uno de sus caracteres a través de un índice entero:

```
string s = "hello";
Console.WriteLine (s[0]); // 'h'
Console.WriteLine (s[3]); // 'l'
```

La sintaxis para usar indexadores es similar a la de los arreglos, excepto que el(los) argumento(s) de índice puede(n) ser de cualquier tipo(s).

Los indexadores tienen los mismos modificadores que las propiedades (ver "Propiedades" en la página 105) y pueden ser llamados de forma condicional a nulo al insertar un signo de interrogación antes del corchete cuadrado (ver "Operadores de Nulo" en la página 74):

```
string s = null;
Console.WriteLine (s?[0]);  // Writes nothing; no error.
```

Implementación de un indexador

Para escribir un indexador, se define una propiedad llamada "this" y se especifican los argumentos entre corchetes cuadrados:

```
class Sentence
{
  string[] words = "The quick brown fox".Split();

  public string this [int wordNum]      // indexer
  {
    get { return words [wordNum];  }
    set { words [wordNum] = value; }
  }
}
```

Así es cómo se puede utilizar este indexador:

```
Sentence s = new Sentence();
Console.WriteLine (s[3]);       // fox
s[3] = "kangaroo";
Console.WriteLine (s[3]);       // kangaroo
```

Un tipo puede declarar varios indexadores, cada uno con parámetros de diferentes tipos. Un indexador también puede tomar más de un parámetro:

```
public string this [int arg1, string arg2]
{
  get { ... }  set { ... }
}
```

Si omites el accesor de escritura, el indexador se convierte en de solo lectura y puedes usar la sintaxis de cuerpo de expresión para acortar su definición:

```
public string this [int wordNum] => words [wordNum];
```

Implementación del CLR para indexadores

Los indexadores se compilan internamente en métodos llamados get_Item y set_Item, de la siguiente manera:

```
public string get_Item (int wordNum) {...}
public void set_Item (int wordNum, string value) {...}
```

Uso de índices y rangos con indexadores:

Puedes admitir índices y rangos (ver "Índices y rangos" en la página 13) en tus propias clases definiendo un indexador con un tipo de parámetro de Index o Range.

Podemos ampliar nuestro ejemplo anterior agregando los siguientes indexadores a la clase Sentence:

```
public string this [Index index] => words [index];
public string[] this [Range range] => words [range];
```

Esto nos permite hacer lo siguiente:

```
Sentence s = new Sentence();
Console.WriteLine (s [^1]);         // fox
string[] firstTwoWords = s [..2];   // (The, quick)
```

Constructores estáticos

Un constructor estático se ejecuta una vez por tipo en lugar de una vez por instancia. Un tipo solo puede definir un constructor estático, y debe ser sin parámetros y tener el mismo nombre que el tipo:

```
class Test
{
  static Test() { Console.WriteLine ("Type Initialized"); }
}
```

El tiempo de ejecución invoca automáticamente un constructor estático justo antes de que se use el tipo. Dos cosas desencadenan esto:

- Instanciar el tipo.
- Acceder a un miembro estático del tipo.

Los únicos modificadores permitidos en los constructores estáticos son `unsafe` y `extern`.

 Si un constructor estático arroja una excepción no controlada, ese tipo se vuelve inutilizable durante toda la vida de la aplicación.

 A partir de C# 9, también puedes definir inicializadores de módulo, que se ejecutan una vez por ensamblado (cuando el ensamblado se carga por primera vez). Para definir un inicializador de módulo, escribe un método static void y luego aplica el atributo [ModuleInitializer] a ese método:

```
[System.Runtime.CompilerServices.ModuleInitializer]
internal static void InitAssembly()
{
  ...
}
```

Constructores estáticos y orden de inicialización de campos

Los inicializadores de campos estáticos se ejecutan justo antes de que se llame al constructor estático. Si un tipo no tiene un constructor estático, los inicializadores de campos estáticos se ejecutarán justo antes de que se utilice el tipo, o en cualquier momento anterior según lo decida el tiempo de ejecución.

Los inicializadores de campos estáticos se ejecutan en el orden en que los campos son declarados. El siguiente ejemplo ilustra esto. X se inicializa en 0, y Y se inicializa en 3:

```
class Foo
{
  public static int X = Y;    // 0
  public static int Y = 3;    // 3
}
```

Si intercambiamos los dos inicializadores de campos, ambos campos se inicializarán en 3. El siguiente ejemplo imprime 0 seguido de 3 porque el inicializador de campo que instancia un objeto Foo se ejecuta antes de que X se inicialice en 3:

```
Console.WriteLine (Foo.X);    // 3

class Foo
{
  public static Foo Instance = new Foo();
  public static int X = 3;

  Foo() => Console.WriteLine (X);    // 0
}
```

Si intercambiamos las dos líneas en negrita, el ejemplo imprimirá 3 seguido de 3.

Clases estáticas

Una clase marcada como estática no puede ser instanciada ni heredada, y debe estar compuesta únicamente por miembros estáticos. Los ejemplos de clases estáticas incluyen las clases System.Console y System.Math.

Finalizadores

Los finalizadores son métodos de solo clase que se ejecutan antes de que el recolector de basura recupere la memoria de un objeto no referenciado. La sintaxis para un finalizador es el nombre de la clase precedido por el símbolo ~:

```
class Class1
{
  ~Class1()
  {
    ...
  }
}
```

v

En realidad, esta es la sintaxis de C# para anular el método Finalize de la clase Object, y el compilador la expande en la siguiente declaración de método:

```
protected override void Finalize()
{
  ...
  base.Finalize();
}
```

Discutiremos la recolección de basura y los finalizadores en detalle en el Capítulo 12.

Los finalizadores permiten el modificador siguiente:

Unmanaged code modifier	`unsafe`

Puedes escribir finalizadores de una sola declaración utilizando la sintaxis de cuerpo de expresión:

```
~Class1() => Console.WriteLine ("Finalizing");
```

Tipos y métodos parciales

Los tipos parciales permiten dividir la definición de un tipo en múltiples archivos, por lo general. Un escenario común es cuando una clase parcial es generada automáticamente desde otra fuente (como una plantilla o diseñador de Visual Studio) y luego se complementa con métodos adicionales escritos manualmente:

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }

// PaymentForm.cs - hand-authored
partial class PaymentForm { ... }
```

Cada participante debe tener la declaración parcial; lo siguiente es incorrecto:

```
partial class PaymentForm {}
class PaymentForm {}
```

Los participantes no pueden tener miembros en conflicto. Por ejemplo, no se puede repetir un constructor con los mismos parámetros. Los tipos parciales se resuelven completamente por el compilador, lo que significa que cada participante debe estar disponible en tiempo de compilación y debe residir en la misma assembly.

Puedes especificar una clase base en una o más declaraciones de clases parciales, siempre que la clase base, si se especifica, sea la misma. Además, cada participante puede especificar interfaces para implementar de forma independiente. Cubrimos las clases base e interfaces en "Herencia" en la página 115 y "Interfaces" en la página 134.

El compilador no garantiza el orden de inicialización de los campos entre las declaraciones de tipos parciales.

Métodos parciales

Un tipo parcial puede contener métodos parciales. Estos permiten que un tipo parcial generado automáticamente proporcione puntos de extensión personalizables para la autoría manual. Por ejemplo:

```
partial class PaymentForm    // In auto-generated file
{
  ...
  partial void ValidatePayment (decimal amount);
}

partial class PaymentForm    // In hand-authored file
{
  ...


  partial void ValidatePayment (decimal amount)
  {
    if (amount > 100)
      ...
  }
}
```

Un método parcial consta de dos partes: una definición y una implementación. La definición generalmente es generada automáticamente por un generador de código, mientras que la implementación es escrita manualmente. Si no se proporciona una implementación, la definición del método parcial se elimina durante la compilación (al igual que el código que lo llama). Esto

permite que el código generado automáticamente sea flexible al proporcionar puntos de extensión sin preocuparse por el exceso de código.

Los métodos parciales deben ser de tipo void y son implícitamente privados. No pueden incluir parámetros de salida (out parameters).

Métodos parciales extendidos

Los métodos parciales extendidos (a partir de C# 9) están diseñados para el escenario de generación de código inverso, donde un programador define puntos de extensión que un generador de código implementa. Un ejemplo de esto podría ser el uso de generadores de código fuente, una característica de Roslyn que permite alimentar al compilador un ensamblado que genera automáticamente partes de tu código.

Una declaración de método parcial se considera extendida si comienza con un modificador de accesibilidad:

```
public partial class Test
{
  public partial void M1();    // Extended partial method
  private partial void M2();   // Extended partial method
}
```

La presencia del modificador de accesibilidad no solo afecta a la accesibilidad, sino que también le indica al compilador que trate la declaración de manera diferente.

Los métodos parciales extendidos deben tener implementaciones; no se eliminan si no se implementan. En este ejemplo, tanto M1 como M2 deben tener implementaciones porque cada uno especifica modificadores de accesibilidad (public y private).

Debido a que no se eliminan, los métodos parciales extendidos pueden devolver cualquier tipo y pueden incluir parámetros de salida (out parameters):

```
public partial class Test
{
  public partial bool IsValid (string identifier);
  internal partial bool TryParse (string number, out int result);
}
```

El operador `nameof`

El operador `nameof` devuelve el nombre de cualquier símbolo (tipo, miembro, variable, etc.) como una cadena:

```
int count = 123;
string name = nameof (count);       // name is "count"
```

Su ventaja sobre simplemente especificar una cadena es la verificación de tipos estática. Herramientas como Visual Studio pueden entender la referencia al símbolo, por lo que si cambias el nombre del símbolo en cuestión, todas sus referencias también se cambiarán de nombre.

Para especificar el nombre de un miembro de tipo, como un campo o una propiedad, incluye también el tipo. Esto funciona tanto con miembros estáticos como de instancia:

```
string name = nameof (StringBuilder.Length);
```

Esto se evalúa como "Length". Para obtener "StringBuilder.Length", harías lo siguiente:

```
nameof (StringBuilder) + "." + nameof (StringBuilder.Length);
```

Herencia

Una clase puede heredar de otra clase para extender o personalizar la clase original. Heredar de una clase te permite reutilizar la funcionalidad de esa clase en lugar de construirla desde cero. Una clase solo puede heredar de una única clase, pero puede ser heredada por muchas clases, formando así una jerarquía de clases. En este ejemplo, comenzamos definiendo una clase llamada "Asset":

```
public class Asset
{
  public string Name;
}
```

A continuación, definimos las clases "Stock" y "House", que heredarán de "Asset". "Stock" y "House" obtienen todo lo que tiene "Asset", además de cualquier miembro adicional que definan:

```
public class Stock : Asset   // inherits from Asset
{
  public long SharesOwned;
}

public class House : Asset   // inherits from Asset
{
  public decimal Mortgage;
}
```

Así es como podemos usar estas clases:

```
Stock msft = new Stock { Name="MSFT",
                         SharesOwned=1000 };

Console.WriteLine (msft.Name);         // MSFT
Console.WriteLine (msft.SharesOwned);  // 1000

House mansion = new House { Name="Mansion",
                            Mortgage=250000 };


Console.WriteLine (mansion.Name);      // Mansion
Console.WriteLine (mansion.Mortgage);  // 250000
```

Las clases derivadas, Stock y House, heredan el campo Name de la clase base Asset.



Una clase derivada también se llama subclase.

Una clase base también se llama superclase.

Polimorfismo

Las referencias son polimórficas. Esto significa que una variable de tipo x puede referirse a un objeto que es una subclase de x. Por ejemplo, considera el siguiente método:

```
public static void Display (Asset asset)
{
  System.Console.WriteLine (asset.Name);
}
```

Este método puede mostrar tanto un Stock como una House porque ambos son Assets:

```
Stock msft    = new Stock ... ;
House mansion = new House ... ;

Display (msft);
Display (mansion);
```

El polimorfismo funciona en base a que las subclases (Stock y House) tienen todas las características de su clase base (Asset). Sin embargo, lo contrario no es cierto. Si se modificara el método Display para aceptar una House, no se podría pasar un objeto Asset:

```
Display (new Asset());     // Compile-time error

public static void Display (House house)         // Will not accept Asset
{
  System.Console.WriteLine (house.Mortgage);
}
```

Casting y conversiones de referencia

Una referencia de objeto puede ser:

- Implícitamente ascendida a una referencia de la clase base.
- Explícitamente descendida a una referencia de la subclase.

El casting hacia arriba y hacia abajo entre tipos de referencia compatibles realiza conversiones de referencia: se crea una nueva referencia (lógicamente) que apunta al mismo objeto. Una conversión hacia arriba siempre tiene éxito; una conversión hacia abajo solo tiene éxito si el objeto tiene el tipo adecuado.

Upcasting

Una operación de upcasting crea una referencia de la clase base a partir de una referencia de la subclase:

```
Stock msft = new Stock();
Asset a = msft;              // Upcast
```

Después del upcasting, la variable "a" sigue haciendo referencia al mismo objeto Stock que la variable "msft". El objeto al que se hace referencia no se altera o convierte en sí mismo:

```
Console.WriteLine (a == msft);        // True
```

Aunque "a" y "msft" hacen referencia al mismo objeto, "a" tiene una visión más restrictiva de ese objeto:

```
Console.WriteLine (a.Name);           // OK
Console.WriteLine (a.SharesOwned);    // Compile-time error
```

La última línea genera un error en tiempo de compilación porque la variable "a" es de tipo Asset, aunque se refiera a un objeto de tipo Stock. Para acceder al campo SharesOwned, debes hacer un downcast del Asset a un Stock.

Downcasting

Una operación de downcasting crea una referencia de la subclase a partir de una referencia de la clase base:

```
Stock msft = new Stock();
Asset a = msft;                       // Upcast
Stock s = (Stock)a;                   // Downcast
Console.WriteLine (s.SharesOwned);    // <No error>
Console.WriteLine (s == a);           // True
Console.WriteLine (s == msft);        // True
```

Como con un upcast, solo las referencias se ven afectadas, no el objeto subyacente. Un downcast requiere una conversión explícita porque puede fallar en tiempo de ejecución:

```
House h = new House();
Asset a = h;              // Upcast always succeeds
Stock s = (Stock)a;       // Downcast fails: a is not a Stock
```

Si un downcast falla, se lanza una excepción InvalidCastException. Esto es un ejemplo de verificación de tipos en tiempo de ejecución (se detalla este concepto en "Verificación de tipos estáticos y en tiempo de ejecución" en la página 127).

El operador "as"

El operador "as" realiza un downcast que evalúa como nulo (en lugar de lanzar una excepción) si el downcast falla:

```
Asset a = new Asset();
Stock s = a as Stock;       // s is null; no exception thrown
```

Esto es útil cuando posteriormente se va a comprobar si el resultado es nulo:

```
if (s != null) Console.WriteLine (s.SharesOwned);
```



Sin una prueba de este tipo, una conversión es ventajosa porque, si falla, se lanza una excepción más útil. Podemos ilustrar esto comparando las siguientes dos líneas de código:

```
long shares = ((Stock)a).SharesOwned;    // Approach #1
long shares = (a as Stock).SharesOwned;  // Approach #2
```

Si "a" no es un objeto Stock, la primera línea lanza una InvalidCastException, que es una descripción precisa de lo que salió mal. La segunda línea lanza una NullReferenceException, que es ambigua. ¿No era "a" un objeto Stock, o era "a" nulo?

Otra forma de verlo es que con el operador de conversión, le estás diciendo al compilador: "Estoy seguro del tipo de valor; si me equivoco, hay un error en mi código, ¡así que lanza una excepción!". Mientras que con el operador "as", no estás seguro de su tipo y quieres tomar una rama según el resultado en tiempo de ejecución.

El operador "as" no puede realizar conversiones personalizadas (consultar "Sobrecarga de operadores" en la página 239), y no puede realizar conversiones numéricas:

```
long x = 3 as long;    // Compile-time error
```



Los operadores "as" y "cast" también pueden realizar conversiones hacia arriba (upcasts), aunque esto no es muy útil ya que una conversión implícita hará el trabajo.

El operador "is"

El operador "is" prueba si una variable coincide con un patrón. C# admite varios tipos de patrones, siendo el más importante el patrón de tipo, donde se sigue la palabra clave "is" seguida del nombre del tipo.

En este contexto, el operador "is" prueba si se podría realizar una conversión de referencia, es decir, si un objeto deriva de una clase especificada (o implementa una interfaz). A menudo se utiliza para realizar pruebas antes de realizar una conversión hacia abajo (downcast):

```
if (a is Stock)
  Console.WriteLine (((Stock)a).SharesOwned);
```

El operador "is" también evalúa como verdadero si se podría realizar una conversión de desempaquetado (unboxing) (consultar "El tipo object" en la página 125). Sin embargo, no considera conversiones personalizadas o numéricas.



El operador "is" funciona con muchos otros patrones introducidos en versiones recientes de C#. Para obtener una discusión completa, consultar "Patrones" en la página 222.

Introducir una variable de patrón

Se puede introducir una variable al utilizar el operador "is":

```
if (a is Stock s)
  Console.WriteLine (s.SharesOwned);
```

Esto es equivalente a lo siguiente:

```
Stock s;
if (a is Stock)
{
  s = (Stock) a;
  Console.WriteLine (s.SharesOwned);
}
```

La variable que introduces está disponible para "consumo" inmediato, por lo que lo siguiente es válido:

```
if (a is Stock s && s.SharesOwned > 100000)
  Console.WriteLine ("Wealthy");
```

Y permanece en ámbito fuera de la expresión 'is', permitiendo lo siguiente:

```
if (a is Stock s && s.SharesOwned > 100000)
  Console.WriteLine ("Wealthy");
Else
  s = new Stock();   // s is in scope

Console.WriteLine (s.SharesOwned);  // Still in scope
```

Miembros de función virtual

Una función marcada como virtual puede ser sobrescrita por subclases que deseen proporcionar una implementación especializada. Los métodos, propiedades, indexadores y eventos pueden declararse como virtuales:

```
public class Asset
{
  public string Name;
  public virtual decimal Liability => 0;   // Expression-bodied property
}
```

(Liability => 0 es un atajo para { get { return 0; } }. Para más detalles sobre esta sintaxis, consulta "Propiedades expresadas en forma de expresión" en la página 107).

Una subclase sobrescribe un método virtual aplicando el modificador override:

```
public class Stock : Asset
{
  public long SharesOwned;
}

public class House : Asset
{
  public decimal Mortgage;
  public override decimal Liability => Mortgage;
}
```

Por defecto, la Liability de un Asset es 0. Un Stock no necesita especializar este comportamiento. Sin embargo, House especializa la propiedad Liability para devolver el valor de la Mortgage:

```
House mansion = new House { Name="McMansion", Mortgage=250000 };
Asset a = mansion;
Console.WriteLine (mansion.Liability);  // 250000
Console.WriteLine (a.Liability);        // 250000
```

Las firmas, tipos de retorno y accesibilidad de los métodos virtuales y sobrescritos deben ser idénticos. Un método sobrescrito puede llamar a la implementación de su clase base mediante la palabra clave base (esto se explica en "La palabra clave base" en la página 122).



Llamar a métodos virtuales desde un constructor es potencialmente peligroso porque los autores de las subclases es probable que no sepan, al sobrescribir el método, que están trabajando con un objeto parcialmente inicializado. En otras palabras, el método sobrescrito podría terminar accediendo a métodos o propiedades que dependen de campos que aún no han sido inicializados por el constructor.

Tipos de retorno covariantes

A partir de C# 9, puedes sobrescribir un método (o un accesor get de una propiedad) de manera que devuelva un tipo más derivado (subclase). Por ejemplo:

```
public class Asset
{
  public string Name;
  public virtual Asset Clone() => new Asset { Name = Name };
}

public class House : Asset
{
  public decimal Mortgage;
  public override House Clone() => new House
                                   { Name = Name, Mortgage = Mortgage };
}
```

Esto está permitido porque no rompe el contrato de que Clone debe devolver un Asset: devuelve un House, que es un Asset (y más).

Antes de C# 9, tenías que sobrescribir métodos con el mismo tipo de retorno:

```
public override Asset Clone() => new House { ... }
```

Esto aún funciona, porque el método Clone sobrescrito instancia un House en lugar de un Asset. Sin embargo, para tratar el objeto devuelto como un House, debes realizar una conversión descendente:

```
House mansion1 = new House { Name="McMansion", Mortgage=250000 };
House mansion2 = (House) mansion1.Clone();
```

Clases abstractas y miembros abstractos

Una clase declarada como abstracta nunca puede ser instanciada. En su lugar, solo se pueden instanciar sus subclases concretas.

Las clases abstractas pueden definir miembros abstractos. Los miembros abstractos son similares a los miembros virtuales, excepto que no proporcionan una implementación por defecto. Esa implementación debe ser proporcionada por la subclase a menos que esa subclase también sea declarada abstracta:

```
public abstract class Asset
{
  // Note empty implementation
  public abstract decimal NetValue { get; }
}

public class Stock : Asset
{
  public long SharesOwned;
  public decimal CurrentPrice;

  // Override like a virtual method.
  public override decimal NetValue => CurrentPrice * SharesOwned;
}
```

Ocultamiento de miembros heredados

Una clase base y una subclase pueden definir miembros idénticos. Por ejemplo:

```
public class A      { public int Counter = 1; }
public class B : A  { public int Counter = 2; }
```

El campo ` Counter ` en la clase B se dice que oculta el campo ` Counter ` en la clase A. Por lo general, esto ocurre por accidente, cuando se agrega un miembro al tipo base después de que se haya agregado un miembro idéntico al subtipo. Por esta razón, el compilador genera una advertencia y luego resuelve la ambigüedad de la siguiente manera:

- Las referencias a A (en tiempo de compilación) se enlazan a A.Counter.
- Las referencias a B (en tiempo de compilación) se enlazan a B.Counter.

Ocasionalmente, se desea ocultar intencionalmente un miembro, en cuyo caso se puede aplicar el modificador `new` al miembro en la subclase. El modificador `new` no hace más que suprimir la advertencia del compilador que de otro modo se generaría:

```
public class A     { public     int Counter = 1; }
public class B : A { public new int Counter = 2; }
```

El modificador `new` comunica tu intención al compilador, y a otros programadores, de que el miembro duplicado no es un accidente.



Correcto, en C# el término "new" se sobrecarga para tener significados independientes en diferentes contextos. Específicamente, el operador "new" es diferente del modificador de miembro "new".

Nuevo versus sobrescribir(New versus override)

Considere la siguiente jerarquía de clases:

```
public class BaseClass
{
  public virtual void Foo()  { Console.WriteLine ("BaseClass.Foo"); }
}

public class Overrider : BaseClass
{
  public override void Foo() { Console.WriteLine ("Overrider.Foo"); }
}

public class Hider : BaseClass
{
  public new void Foo()      { Console.WriteLine ("Hider.Foo"); }
}
```

Las diferencias de comportamiento entre `Overrider` y `Hider` se demuestran en el siguiente código:

```
Overrider over = new Overrider();
BaseClass b1 = over;
over.Foo();                          // Overrider.Foo
b1.Foo();                            // Overrider.Foo

Hider h = new Hider();
BaseClass b2 = h;
h.Foo();                            // Hider.Foo
b2.Foo();                           // BaseClass.Foo
```

Sellando funciones y clases:

Una función miembro anulada puede sellar su implementación con la palabra clave `sealed` para evitar que sea anulada por subclases posteriores. En nuestro ejemplo anterior de función miembro virtual, podríamos haber sellado la implementación de `Liability` en la clase `House` para evitar que una clase derivada de `House` anule `Liability`, de la siguiente manera:

```
public sealed override decimal Liability { get { return Mortgage; } }
```

También puedes aplicar el modificador `sealed` a la propia clase para evitar la subclase. Sellar una clase es más común que sellar una función miembro.

Aunque puedes sellar una función miembro para evitar la anulación, no puedes sellar un miembro para evitar que sea ocultado.

La palabra clave `base`

La palabra clave `base` es similar a la palabra clave `this`. Sirve para dos propósitos esenciales:

- Acceder a una función miembro anulada desde la subclase.
- Llamar al constructor de la clase base (ver la siguiente sección).

En este ejemplo, la clase `House` utiliza la palabra clave `base` para acceder a la implementación de `Liability` de la clase `Asset`:

```
public class House : Asset
{
  ...
  public override decimal Liability => base.Liability + Mortgage;
}
```

Con la palabra clave `base`, accedemos a la propiedad `Liability` de `Asset` de manera no virtual. Esto significa que siempre accederemos a la versión de `Asset` de esta propiedad, independientemente del tipo de tiempo de ejecución real de la instancia.

El mismo enfoque funciona si `Liability` está oculto en lugar de anulado. (También puedes acceder a miembros ocultos mediante una conversión a la clase base antes de invocar la función.)

Constructores e herencia

Una subclase debe declarar sus propios constructores. Los constructores de la clase base son accesibles para la clase derivada, pero no se heredan automáticamente. Por ejemplo, si definimos las clases Baseclass y Subclass de la siguiente manera:

```
public class Baseclass
{
  public int X;
  public Baseclass () { }
  public Baseclass (int x) { this.X = x; }
}

public class Subclass : Baseclass { }
```

El siguiente código es incorrecto:

```
Subclass s = new Subclass (123);
```

Subclass debe "redefinir" cualquier constructor que desee exponer. Sin embargo, al hacerlo, puede llamar a cualquiera de los constructores de la clase base utilizando la palabra clave `base`:

```
public class Subclass : Baseclass
{
  public Subclass (int x) : base (x) { }
}
```

La palabra clave `base` funciona de manera similar a la palabra clave `this`, excepto que llama a un constructor en la clase base.

Llamada implícita al constructor sin parámetros de la clase base

Si un constructor en una subclase omite la palabra clave `base`, se llama implícitamente al constructor sin parámetros del tipo base:

```
public class BaseClass
{
  public int X;
  public BaseClass() { X = 1; }
}

public class Subclass : BaseClass
{
  public Subclass() { Console.WriteLine (X); }  // 1
}
```

Si la clase base no tiene un constructor sin parámetros accesible, las subclases están obligadas a usar la palabra clave `base` en sus constructores.

Orden de inicialización de constructores y campos

Cuando se instancia un objeto, la inicialización se realiza en el siguiente orden:

1. De la subclase a la clase base:
   a. Los campos se inicializan.
   b. Se evalúan los argumentos para las llamadas a los constructores de la clase base.
2. De la clase base a la subclase:
   a. Se ejecutan los cuerpos de los constructores.

Por ejemplo:

```
public class B
{
  int x = 1;          // Executes 3rd
  public B (int x)
  {
    ...               // Executes 4th
  }
}
public class D : B
{
  int y = 1;          // Executes 1st
  public D (int x)
    : base (x + 1)    // Executes 2nd
  {
     ...              // Executes 5th
  }
}
```

Sobrecarga y resolución de métodos

La herencia tiene un impacto interesante en la sobrecarga de métodos. Considera las siguientes dos sobrecargas:

```
static void Foo (Asset a) { }
static void Foo (House h) { }
```

Cuando se llama a una sobrecarga, el tipo más específico tiene prioridad:

```
House h = new House (...);
Foo(h);                       // Calls Foo(House)
```

La sobrecarga específica a llamar se determina estáticamente (en tiempo de compilación) en lugar de en tiempo de ejecución. El siguiente código llama a Foo(Asset), aunque el tipo de tiempo de ejecución de a sea House:

```
Asset a = new House (...);
Foo(a);                       // Calls Foo(Asset)
```



Si conviertes Asset a dynamic (Capítulo 4), la decisión sobre qué sobrecarga llamar se pospone hasta el tiempo de ejecución y se basa en el tipo real del objeto:

```
Asset a = new House (...);
Foo ((dynamic)a);   // Calls Foo(House)
```

El tipo `object`

El tipo `object` (System.Object) es la clase base definitiva para todos los tipos. Cualquier tipo puede ser convertido al tipo `object`.

Para ilustrar cómo esto es útil, considera una pila de propósito general. Una pila es una estructura de datos basada en el principio de LIFO, que significa "último en entrar, primero en salir". Una pila tiene dos operaciones: empujar un objeto en la pila y sacar un objeto de la pila. Aquí tienes una implementación simple que puede contener hasta 10 objetos:

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj)   { data[position++] = obj;  }
  public object Pop()             { return data[--position]; }
}
```

Debido a que la clase `Stack` trabaja con el tipo `object`, podemos empujar (Push) y sacar (Pop) instancias de cualquier tipo hacia y desde la pila:

```
Stack stack = new Stack();
stack.Push ("sausage");
string s = (string) stack.Pop();   // Downcast, so explicit cast is needed

Console.WriteLine (s);             // sausage
```

El tipo `object` es un tipo de referencia debido a que es una clase. A pesar de esto, los tipos de valor, como `int`, también pueden ser convertidos hacia y desde `object`, y así ser añadidos a nuestra pila. Esta característica de C# se conoce como unificación de tipos y se demuestra aquí:

```
stack.Push (3);
int three = (int) stack.Pop();
```

Cuando realizas una conversión entre un tipo de valor y `object`, el CLR debe realizar un trabajo especial para bridar la diferencia en la semántica entre los tipos de valor y los tipos de referencia. Este proceso se llama enboxing (boxing) y desenboxing (unboxing).

 En "Genéricos" en la página 145, describimos cómo mejorar nuestra clase Stack para manejar mejor pilas con elementos del mismo tipo.

Boxing and Unboxing

El enboxing (boxing) es el acto de convertir una instancia de tipo de valor en una instancia de tipo de referencia. El tipo de referencia puede ser la clase `object` o una interfaz (que se verá más adelante en el capítulo). En este ejemplo, realizamos el enboxing de un `int` en un `object`:

```
int x = 9;
object obj = x;           // Box the int
```

El desenboxing (unboxing) revierte la operación al convertir el objeto de vuelta al tipo de valor original:

```
int y = (int)obj;         // Unbox the int
```

El desenboxing requiere una conversión explícita. El tiempo de ejecución verifica que el tipo de valor declarado coincida con el tipo de objeto real y lanza una excepción `InvalidCastException` si la verificación falla. Por ejemplo, lo siguiente lanza una excepción porque `long` no coincide exactamente con `int`:

```
object obj = 9;           // 9 is inferred to be of type int
long x = (long) obj;      // InvalidCastException
```

Sin embargo, lo siguiente tiene éxito:

```
object obj = 9;
long x = (int) obj;
```

Lo mismo sucede con esto:

```
object obj = 3.5;             // 3.5 is inferred to be of type double
int x = (int) (double) obj;   // x is now 3
```

En el último ejemplo, `(double)` realiza un desenboxing y luego `(int)` realiza una conversión numérica.

 Las conversiones de enboxing(boxing) son cruciales para proporcionar un sistema de tipos unificado. Sin embargo, el sistema no es perfecto: veremos en "Genéricos" en la página

145 que la covarianza con matrices (arrays) y genéricos solo admite conversiones de referencia y no conversiones de enboxing.

```
object[] a1 = new string[3];   // Legal
object[] a2 = new int[3];      // Error
```

Copiando la semántica de enboxing(boxing) y desenboxing(unboxing).

El enboxing (boxing) copia la instancia del tipo de valor en el nuevo objeto, y el desenboxing (unboxing) copia el contenido del objeto de vuelta a una instancia del tipo de valor. En el siguiente ejemplo, cambiar el valor de `i` no afecta a su copia enboxed anteriormente:

```
int i = 3;
object boxed = i;
i = 5;
Console.WriteLine (boxed);    // 3
```

Verificación de tipos estática y en tiempo de ejecución

Los programas en C# se verifican en cuanto a tipos tanto estáticamente (en tiempo de compilación) como en tiempo de ejecución (por el CLR).

La verificación de tipos estática permite que el compilador verifique la corrección de tu programa sin ejecutarlo. El siguiente código fallará porque el compilador hace cumplir la tipificación estática:

```
int x = "5";
```

La verificación de tipos en tiempo de ejecución se realiza por el CLR cuando haces una conversión descendente (downcast) mediante una conversión de referencia o unboxing:

```
object y = "5";
int z = (int) y;          // Runtime error, downcast failed
```

La verificación de tipos en tiempo de ejecución es posible porque cada objeto en el montón (heap) almacena internamente un pequeño token de tipo. Puedes obtener este token llamando al método `GetType` del objeto.

El método `GetType` y el operador `typeof`

Todos los tipos en C# se representan en tiempo de ejecución con una instancia de `System.Type`. Hay dos formas básicas de obtener un objeto `System.Type`:

- Llamar a `GetType` en la instancia.
- Usar el operador `typeof` seguido del nombre del tipo.

`GetType` se evalúa en tiempo de ejecución; `typeof` se evalúa estáticamente en tiempo de compilación (cuando se involucran parámetros de tipo genérico, se resuelve por el compilador JIT).

La clase `System.Type` tiene propiedades para obtener información sobre el tipo, como el nombre del tipo, el ensamblado, el tipo base, entre otros:

```
Point p = new Point();
Console.WriteLine (p.GetType().Name);             // Point
Console.WriteLine (typeof (Point).Name);          // Point
Console.WriteLine (p.GetType() == typeof(Point)); // True
Console.WriteLine (p.X.GetType().Name);           // Int32
Console.WriteLine (p.Y.GetType().FullName);       // System.Int32

public class Point { public int X, Y; }
```

`System.Type` también tiene métodos que actúan como una puerta de acceso al modelo de reflexión del tiempo de ejecución, que se describe en el Capítulo 18.

El método `ToString`

El método `ToString` devuelve la representación textual predeterminada de una instancia del tipo. Este método está sobrescrito por todos los tipos incorporados. Aquí hay un ejemplo del uso del método `ToString` del tipo `int`:

```
int x = 1;
string s = x.ToString();     // s is "1"
```

Puedes sobrescribir el método `ToString` en tipos personalizados de la siguiente manera:

```
Panda p = new Panda { Name = "Petey" };
Console.WriteLine (p);   // Petey

public class Panda
{
  public string Name;
  public override string ToString() => Name;
}
```

Si no sobrescribes `ToString`, el método devuelve el nombre del tipo.



Cuando llamas directamente a un miembro de objeto sobrescrito, como `ToString`, en un tipo de valor, no se produce el boxing. El boxing solo ocurre si realizas una conversión explícita:

```
int x = 1;
string s1 = x.ToString();    // Calling on nonboxed value
object box = x;
string s2 = box.ToString();  // Calling on boxed value
```

Listado de miembros de objeto

Aquí están todos los miembros del objeto:

```
public class Object
{
  public Object();

  public extern Type GetType();


   public virtual bool Equals (object obj);
   public static bool Equals  (object objA, object objB);
   public static bool ReferenceEquals (object objA, object objB);

   public virtual int GetHashCode();

   public virtual string ToString();

   protected virtual void Finalize();
   protected extern object MemberwiseClone();
 }
```

Describimos los métodos Equals, ReferenceEquals y GetHashCode en "Comparación de Igualdad" en la página 324.

Estructuras (Structs)

Una estructura es similar a una clase, con las siguientes diferencias clave:

- Una estructura es un tipo de valor, mientras que una clase es un tipo de referencia.
- Una estructura no admite la herencia (excepto por la herencia implícita de object, o más precisamente, System.ValueType).

Una estructura puede tener todos los miembros que puede tener una clase, excepto por un finalizador. Y debido a que no se puede subclasificar, los miembros no pueden marcarse como virtuales, abstractos o protegidos.

 Antes de C# 10, las estructuras tenían prohibido definir inicializadores de campos y constructores sin parámetros. Aunque esta prohibición se ha relajado ahora, principalmente para beneficiar a las estructuras de tipo registro (ver "Registros" en la página 211), vale la pena pensarlo detenidamente antes de definir estos constructos, ya que pueden resultar en un comportamiento confuso que describiremos en "Semántica de Construcción de Estructuras" en la página 129.

Una estructura es apropiada cuando se desean semánticas de tipo de valor. Buenos ejemplos de estructuras son los tipos numéricos, donde es más natural que la asignación copie un valor en lugar de una referencia. Debido a que una estructura es un tipo de valor, cada instancia no requiere la creación de un objeto en el montón de memoria; esto resulta en ahorros útiles al crear muchas instancias de un tipo. Por ejemplo, crear una matriz de elementos de tipo de valor requiere solo una asignación en el montón de memoria.

Debido a que las estructuras son tipos de valor, una instancia no puede ser nula. El valor predeterminado para una estructura es una instancia vacía, con todos los campos vacíos (establecidos en sus valores predeterminados).

Semántica de Construcción de Estructuras

A diferencia de las clases, cada campo en una estructura debe asignarse explícitamente en el constructor (o en el inicializador de campo). Por ejemplo:

```
struct Point
{
  int x, y;
  public Point (int x, int y) { this.x = x; this.y = y; }  // OK
}
```

Si agregamos el siguiente constructor, la estructura no se compilaría porque "y" quedaría sin asignar:

```
public Point (int x)        { this.x = x; }  // Not OK
```

El constructor predeterminado

Además de cualquier constructor que definas, una estructura siempre tiene un constructor implícito sin parámetros que realiza una operación de inicialización a cero de sus campos (estableciéndolos en sus valores predeterminados):

```
Point p = new Point();        // p.x and p.y will be 0
struct Point { int x, y; }
```

Incluso cuando defines un constructor sin parámetros propio, el constructor implícito sin parámetros aún existe y se puede acceder mediante la palabra clave "default":

```
Point p1 = new Point();       // p1.x and p1.y will be 1
Point p2 = default;           // p2.x and p2.y will be 0

struct Point
{
  int x = 1;
  int y;
  public Point() => y = 1;
}
```

En este ejemplo, inicializamos "x" a 1 mediante un inicializador de campo, e inicializamos "y" a 1 mediante el constructor sin parámetros. Sin embargo, con la palabra clave "default", aún pudimos crear un Point que omitió ambas inicializaciones. El constructor predeterminado también se puede acceder de otras formas, como ilustra el siguiente ejemplo:

```
var points = new Point[10];   // Each point in the array will be (0,0)
var test = new Test();        // test.p will be (0,0)

class Test { Point p; }
```



Tener lo que equivale a dos constructores sin parámetros puede ser una fuente de confusión y es una buena razón para evitar definir inicializadores de campo y constructores explícitos sin parámetros en estructuras.

Una buena estrategia con las estructuras es diseñarlas de manera que su valor predeterminado sea un estado válido, lo que hace que la inicialización sea redundante. Por ejemplo, en lugar de esto:

```
struct WebOptions { public string Protocol { get; set; } = "https"; }
```

considera lo siguiente:

```
struct WebOptions
{
  string protocol;
  public string Protocol { get => protocol ?? "https";
                           set => protocol = value;    }
}
```

Estructuras y funciones de solo lectura.

Puedes aplicar el modificador "readonly" a una estructura (struct) para garantizar que todos los campos sean de solo lectura; esto ayuda a declarar la intención y también brinda mayor libertad de optimización al compilador. Por ejemplo:

```
readonly struct Point
{
  public readonly int X, Y;   // X and Y must be readonly
}
```

Si necesitas aplicar "readonly" a un nivel más granular, puedes aplicar el modificador "readonly" (a partir de C# 8) a las funciones de la estructura. Esto garantiza que, si la función intenta modificar algún campo, se genere un error en tiempo de compilación. Por ejemplo:

```
struct Point
{
  public int X, Y;
  public readonly void ResetX() => X = 0;  // Error!
}
```

Si una función de solo lectura llama a una función no readonly, el compilador generará una advertencia (y copiará la estructura de manera defensiva para evitar la posibilidad de una mutación).

Estructuras ref



Las estructuras de referencia (ref structs) se introdujeron en C# 7.2 como una característica especializada principalmente para beneficiar a las estructuras Span<T> y

ReadOnlySpan<T> que se describen en el Capítulo 23 (y a la altamente optimizada Utf8JsonReader que se describe en el Capítulo 11). Estas estructuras ayudan en una técnica de micro-optimización que tiene como objetivo reducir las asignaciones de memoria.

A diferencia de los tipos de referencia, cuyas instancias siempre se alojan en el montón (heap), los tipos de valor viven en el lugar donde se declaró la variable. Si un tipo de valor aparece como parámetro o variable local, residirá en la pila (stack).

```
void SomeMethod()
{
  Point p;   // p will reside on the stack
}
struct Point { public int X, Y; }
```

Pero si un tipo de valor aparece como campo en una clase, residirá en el montón (heap).

```
class MyClass
{
  Point p;   // Lives on heap, because MyClass instances live on the heap
}
```

Del mismo modo, los arreglos de estructuras residen en el montón (heap), y al convertir una estructura en un objeto (boxing), se envía al montón. Agregar el modificador ref a la declaración de una estructura garantiza que solo pueda residir en la pila. Intentar utilizar una ref struct de manera que pueda residir en el montón generará un error en tiempo de compilación:

```
var points = new Point [100];           // Error: will not compile!

ref struct Point { public int X, Y; }
class MyClass    { Point P;         }   // Error: will not compile!
```

Las ref structs se introdujeron principalmente para beneficiar a las estructuras Span<T> y ReadOnlySpan<T>. Dado que las instancias de Span<T> y ReadOnlySpan<T> solo pueden existir en la pila, es posible que envuelvan de manera segura memoria asignada en la pila.

Las ref structs no pueden participar en ninguna característica de C# que introduzca directa o indirectamente la posibilidad de existir en el montón. Esto incluye varias características avanzadas de C# que describimos en el Capítulo 4, como expresiones lambda, iteradores y funciones asíncronas (porque, detrás de escena, estas características crean clases ocultas con campos). Además, las ref structs no pueden aparecer dentro de structs no ref y no pueden implementar interfaces (porque esto podría resultar en boxing).

Modificadores de Acceso

Para promover la encapsulación, un tipo o miembro de tipo puede limitar su accesibilidad a otros tipos y otras bibliotecas al agregar uno de los cinco modificadores de acceso a la declaración:

1. public

Totalmente accesible. Esta es la accesibilidad implícita para los miembros de una enumeración (enum) o interfaz.

2. internal

Accesible solo dentro de la biblioteca que lo contiene o en bibliotecas amigas. Esta es la accesibilidad predeterminada para los tipos no anidados.

3. private

Accesible solo dentro del tipo que lo contiene. Esta es la accesibilidad predeterminada para los miembros de una clase o estructura.

4. protected

Accesible solo dentro del tipo que lo contiene o en subclases.

5. protected internal

Unión de las accesibilidades protected e internal. Un miembro que es protected internal es accesible de dos formas.

6. private protected (a partir de C# 7.2)

Intersección de las accesibilidades protected e internal. Un miembro que es private protected solo es accesible dentro del tipo que lo contiene o en subclases que residen en la misma biblioteca (lo que lo hace menos accesible que protected o internal por separado).

Ejemplos:

La clase Class2 es accesible desde fuera de su ensamblado; la clase Class1 no lo es:

```
class Class1 {}                  // Class1 is internal (default)
public class Class2 {}
```

La clase ClassB expone el campo x a otros tipos en el mismo ensamblado; la clase ClassA no lo hace:

```
class ClassA { int x;          } // x is private (default)
class ClassB { internal int x; }
```

Las funciones dentro de la clase Subclass pueden llamar a Bar pero no a Foo:

```
class BaseClass
{
  void Foo()           {}        // Foo is private (default)
  protected void Bar() {}
}

class Subclass : BaseClass
{
  void Test1() { Foo(); }       // Error - cannot access Foo
  void Test2() { Bar(); }       // OK
}
```

Bibliotecas amigas(Friend Assemblies)

Puedes exponer miembros internos a otras bibliotecas amigas agregando el atributo de ensamblado System.Runtime.CompilerServices.InternalsVisibleTo, especificando el nombre de la biblioteca amiga de la siguiente manera:

```
[assembly: InternalsVisibleTo ("Friend")]
```

Si la biblioteca amiga tiene un nombre fuerte (ver Capítulo 17), debes especificar su clave pública completa de 160 bytes:

```
[assembly: InternalsVisibleTo ("StrongFriend, PublicKey=0024f000048c...")]
```

Puedes extraer la clave pública completa de una biblioteca fuertemente nombrada con una consulta LINQ (explicamos LINQ en detalle en el Capítulo 8):

```
string key = string.Join ("",
  Assembly.GetExecutingAssembly().GetName().GetPublicKey()
    .Select (b => b.ToString ("x2")));
```



El ejemplo de compañía en LINQPad te invita a buscar un ensamblado y luego copia la clave pública completa del ensamblado al portapapeles.

Limitación de accesibilidad(Accessibility Capping)

La accesibilidad de los miembros declarados de un tipo está limitada por dicho tipo. Un ejemplo común de limitación se da cuando tienes un tipo interno con miembros públicos. Por ejemplo, considera lo siguiente:

```
class C { public void Foo() {} }
```

La accesibilidad interna (por defecto) de C limita la accesibilidad de Foo, efectivamente haciendo que Foo sea interno. Una razón común para marcar Foo como público es facilitar la refactorización en caso de que C se cambie posteriormente a público.

Restricciones en los modificadores de acceso

Cuando se anula una función de una clase base, la accesibilidad debe ser idéntica a la de la función anulada. Por ejemplo:

```
class BaseClass             { protected virtual  void Foo() {} }
class Subclass1 : BaseClass { protected override void Foo() {} }  // OK
class Subclass2 : BaseClass { public    override void Foo() {} }  // Error
```

(Existe una excepción cuando se anula un método protegido interno en otro ensamblado, en cuyo caso la anulación simplemente debe ser protegida).

El compilador evita cualquier uso inconsistente de los modificadores de acceso. Por ejemplo, una subclase puede tener una accesibilidad menor que la de una clase base, pero no mayor:

```
internal class A {}
public class B : A {}          // Error
```

Interfaces

Una interfaz es similar a una clase, pero solo especifica comportamiento y no almacena datos. En consecuencia:

- Una interfaz solo puede definir funciones y no campos.
- Los miembros de interfaz son implícitamente abstractos. (Aunque las funciones no abstractas están permitidas desde C# 8, esto se considera un caso especial, que describimos en "Miembros de interfaz predeterminados" en la página 139).
- Una clase (o struct) puede implementar múltiples interfaces. En contraste, una clase solo puede heredar de una sola clase, y una struct no puede heredar en absoluto (aparte de derivar de System.ValueType).

Una declaración de interfaz es similar a una declaración de clase, pero (normalmente) no proporciona implementación para sus miembros porque sus miembros son implícitamente abstractos. Estos miembros serán implementados por las clases y estructuras que implementen la interfaz. Una interfaz solo puede contener funciones, es decir, métodos, propiedades, eventos e indexadores (que curiosamente son precisamente los miembros de una clase que pueden ser abstractos).

Como se define en System.Collections, aquí está la definición de la interfaz IEnumerator:

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

Los miembros de la interfaz siempre son implícitamente públicos y no pueden declarar un modificador de acceso. Implementar una interfaz significa proporcionar una implementación pública para todos sus miembros:

```
internal class Countdown : IEnumerator
{
  int count = 11;
  public bool MoveNext() => count-- > 0;
  public object Current => count;
  public void Reset() { throw new NotSupportedException(); }
}
```

Puedes convertir implícitamente un objeto a cualquier interfaz que implemente:

```
IEnumerator e = new Countdown();
while (e.MoveNext())
  Console.Write (e.Current);      // 109876543210
```



Aunque Countdown sea una clase interna, sus miembros que implementan IEnumerator pueden ser llamados públicamente al convertir una instancia de Countdown a IEnumerator. Por ejemplo, si un tipo público en la misma assembly define un método de la siguiente manera:

```
public static class Util
{
  public static object GetCountDown() => new CountDown();
}
```

Un llamador desde otra assembly podría hacer esto:

```
IEnumerator e = (IEnumerator) Util.GetCountDown();
e.MoveNext();
```

Si IEnumerator se definiera como internal, esto no sería posible.

Extender una interfaz

Las interfaces pueden derivar de otras interfaces. Por ejemplo:

```
public interface IUndoable             { void Undo(); }
public interface IRedoable : IUndoable { void Redo(); }
```

La interfaz IRedoable "hereda" todos los miembros de IUndoable. En otras palabras, los tipos que implementan IRedoable también deben implementar los miembros de IUndoable.

Implementación explícita de interfaces

Al implementar múltiples interfaces, a veces puede haber colisiones entre las firmas de los miembros. Puedes resolver esas colisiones mediante la implementación explícita de un miembro de la interfaz. Considera el siguiente ejemplo:

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget : I1, I2
{
  public void Foo()
  {
    Console.WriteLine ("Widget's implementation of I1.Foo");
  }

  int I2.Foo()
  {
    Console.WriteLine ("Widget's implementation of I2.Foo");
    return 42;
  }
}
```

Dado que I1 e I2 tienen firmas conflictivas para el método Foo, Widget implementa explícitamente el método Foo de I2. Esto permite que los dos métodos coexistan en una sola clase. La única forma de llamar a un miembro implementado explícitamente es hacer una conversión al tipo de interfaz correspondiente:

```
Widget w = new Widget();
w.Foo();                    // Widget's implementation of I1.Foo
((I1)w).Foo();              // Widget's implementation of I1.Foo
((I2)w).Foo();              // Widget's implementation of I2.Foo
```

Otra razón para implementar explícitamente los miembros de una interfaz es ocultar aquellos miembros que son altamente especializados y que podrían resultar distractivos para el uso normal de un tipo. Por ejemplo, un tipo que implementa ISerializable normalmente desearía evitar mostrar sus miembros de ISerializable a menos que se realice una conversión explícita a esa interfaz.

Implementar miembros de una interfaz de forma virtual

Un miembro de interfaz implementado implícitamente está, por defecto, sellado (sealed). Debe ser marcado como virtual o abstract en la clase base para poder ser sobreescrito (overridden):

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
  public virtual void Undo() => Console.WriteLine ("TextBox.Undo");
}

public class RichTextBox : TextBox
{
  public override void Undo() => Console.WriteLine ("RichTextBox.Undo");
}
```

Llamar al miembro de la interfaz a través de la clase base o de la interfaz llama a la implementación de la subclase:

```
RichTextBox r = new RichTextBox();
r.Undo();                       // RichTextBox.Undo
((IUndoable)r).Undo();          // RichTextBox.Undo
((TextBox)r).Undo();            // RichTextBox.Undo
```

Un miembro de una interfaz implementado explícitamente no puede ser marcado como virtual, ni puede ser sobrescrito de la manera habitual. Sin embargo, puede ser reimplantado.

Reimplementar una interfaz en una subclase

Una subclase puede reimplementar cualquier miembro de interfaz ya implementado por una clase base. La reimplantación "secuestra" una implementación de un miembro (cuando se llama a través de la interfaz) y funciona tanto si el miembro es virtual en la clase base como si no lo es.

También funciona tanto si el miembro se implementa implícitamente como si se implementa explícitamente, aunque funciona mejor en este último caso, como demostraremos.

En el siguiente ejemplo, TextBox implementa IUndoable.Undo de forma explícita, por lo que no puede ser marcado como virtual. Para "sobrescribirlo", RichTextBox debe reimplementar el método Undo de IUndoable:

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
  void IUndoable.Undo() => Console.WriteLine ("TextBox.Undo");
}

public class RichTextBox : TextBox, IUndoable
{
  public void Undo() => Console.WriteLine ("RichTextBox.Undo");
}
```

Llamar al miembro reimplementado a través de la interfaz llama a la implementación de la subclase:

```
RichTextBox r = new RichTextBox();
r.Undo();                 // RichTextBox.Undo      Case 1
((IUndoable)r).Undo();    // RichTextBox.Undo      Case 2
```

Supongamos que, con la misma definición de RichTextBox, TextBox implementa Undo de forma implícita:

```
public class TextBox : IUndoable
{
  public void Undo() => Console.WriteLine ("TextBox.Undo");
}
```

Esto nos daría otra forma de llamar a Undo, lo cual "rompería" el sistema, como se muestra en el Caso 3:

```
RichTextBox r = new RichTextBox();
r.Undo();                 // RichTextBox.Undo      Case 1


((IUndoable)r).Undo();    // RichTextBox.Undo      Case 2
((TextBox)r).Undo();      // TextBox.Undo          Case 3
```

El caso 3 demuestra que la reimplantación solo es efectiva cuando se llama a un miembro a través de la interfaz y no a través de la clase base. Esto suele ser indeseable ya que puede crear semánticas inconsistentes. Por lo tanto, la reimplantación es más apropiada como una estrategia para sobrescribir miembros de interfaz implementados explícitamente.

Alternativas a la reimplantación de interfaces

Incluso con la implementación explícita de miembros, la reimplantación de interfaces puede tener problemas por las siguientes razones:

- La subclase no tiene forma de llamar al método de la clase base.
- El autor de la clase base puede no anticipar que se realizará una reimplantación y no permitir las consecuencias potenciales.

La reimplantación puede ser una última opción cuando no se ha previsto la subclase. Sin embargo, una mejor opción es diseñar una clase base de manera que la reimplantación nunca sea necesaria. Hay dos formas de lograr esto:

- Cuando se implementa un miembro de forma implícita, marcarlo como virtual si es apropiado.
- Cuando se implementa un miembro de forma explícita, utilizar el siguiente patrón si se anticipa que las subclases pueden necesitar sobrescribir la lógica:

```
public class TextBox : IUndoable
{
  void IUndoable.Undo()         => Undo();    // Calls method below
  protected virtual void Undo() => Console.WriteLine ("TextBox.Undo");
}

public class RichTextBox : TextBox
{
  protected override void Undo() => Console.WriteLine("RichTextBox.Undo");
}
```

Si no se anticipa ninguna subclasificación, se puede marcar la clase como "sealed" para evitar la reimplantación de interfaces.

Interfaces y boxing

La conversión de una estructura (struct) a una interfaz causa boxing. Llamar a un miembro implementado implícitamente en una estructura no causa boxing:

```
interface  I { void Foo();          }
struct S : I { public void Foo() {} }

...
S s = new S();
s.Foo();         // No boxing.


I i = s;         // Box occurs when casting to interface.
i.Foo();
```

Miembros predeterminados de una interfaz

A partir de C# 8, puedes agregar una implementación predeterminada a un miembro de interfaz, lo que lo hace opcional de implementar:

```
interface ILogger
{
  void Log (string text) => Console.WriteLine (text);
}
```

Esto es ventajoso si deseas agregar un miembro a una interfaz definida en una biblioteca popular sin romper (potencialmente miles de) implementaciones.

Las implementaciones predeterminadas siempre son explícitas, por lo que si una clase que implementa ILogger no define un método Log, la única forma de llamarlo es a través de la interfaz:

```
class Logger : ILogger { }
...
((ILogger)new Logger()).Log ("message");
```

Esto evita un problema de herencia de implementación múltiple: si se agrega el mismo miembro predeterminado a dos interfaces que una clase implementa, nunca habrá ambigüedad sobre qué miembro se llama.

Las interfaces también pueden definir miembros estáticos (incluyendo campos), que pueden ser accedidos desde el código dentro de las implementaciones predeterminadas:

```
interface ILogger
{
  void Log (string text) =>
    Console.WriteLine (Prefix + text);

  static string Prefix = "";
}
```

Dado que los miembros de interfaz son implícitamente públicos, también puedes acceder a los miembros estáticos desde el exterior:

```
ILogger.Prefix = "File log: ";
```

Puedes restringir esto agregando un modificador de accesibilidad al miembro estático de la interfaz (como private, protected o internal).

Los campos de instancia siguen estando prohibidos. Esto está en línea con el principio de las interfaces, que es definir comportamiento y no estado.

Escribir una Clase versus una Interfaz

Como guía:

• Utiliza clases y subclases para tipos que comparten naturalmente una implementación.

• Utiliza interfaces para tipos que tienen implementaciones independientes.

Considera las siguientes clases:

```
abstract class Animal {}
abstract class Bird           : Animal {}
abstract class Insect         : Animal {}
abstract class FlyingCreature : Animal {}
abstract class Carnivore      : Animal {}

// Concrete classes:

class Ostrich : Bird {}
class Eagle   : Bird, FlyingCreature, Carnivore {}  // Illegal
class Bee     : Insect, FlyingCreature {}           // Illegal
class Flea    : Insect, Carnivore {}                // Illegal
```

Las clases Eagle, Bee y Flea no se pueden compilar porque heredar de múltiples clases está prohibido. Para resolver esto, debemos convertir algunos de los tipos en interfaces. Entonces surge la pregunta, ¿qué tipos? Siguiendo nuestra regla general, podríamos decir que los insectos comparten una implementación, al igual que las aves, por lo que seguirán siendo clases. En cambio, las criaturas voladoras tienen mecanismos independientes para volar y los carnívoros tienen estrategias independientes para comer animales, por lo que convertiremos FlyingCreature y Carnivore en interfaces:

```
interface IFlyingCreature {}
interface ICarnivore     {}
```

En un escenario típico, Bird e Insect podrían corresponder a un control de Windows y un control web; FlyingCreature y Carnivore podrían corresponder a IPrintable e IUndoable.

Enumeraciones

Las enumeraciones (enums) son un tipo de valor especial que te permite especificar un grupo de constantes numéricas con nombres. Por ejemplo:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Podemos utilizar este tipo de enumeración de la siguiente manera:

```
BorderSide topSide = BorderSide.Top;
bool isTop = (topSide == BorderSide.Top);   // true
```

Cada miembro de la enumeración tiene un valor numérico subyacente. Por defecto, estos valores son:

- Los valores subyacentes son del tipo int.
- Las constantes 0, 1, 2... se asignan automáticamente en el orden de declaración de los miembros de la enumeración.

Puedes especificar un tipo integral alternativo de la siguiente manera:

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

También puedes especificar un valor subyacente explícito para cada miembro de la enumeración:

```
public enum BorderSide : byte { Left=1, Right=2, Top=10, Bottom=11 }
```



El compilador también te permite asignar explícitamente algunos de los miembros de la enumeración. Los miembros de la enumeración no asignados se incrementan a partir del último valor explícito. El ejemplo anterior es equivalente al siguiente:

```
public enum BorderSide : byte
 { Left=1, Right, Top=10, Bottom }
```

Conversiones de enumeraciones

Puedes convertir una instancia de una enumeración hacia y desde su valor integral subyacente mediante una conversión explícita:

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
bool leftOrRight = (int) side <= 2;
```

También puedes realizar una conversión explícita de un tipo de enumeración a otro. Supongamos que `HorizontalAlignment` está definido de la siguiente manera:

```
public enum HorizontalAlignment
{
  Left = BorderSide.Left,
  Right = BorderSide.Right,
  Center
}
```

Una traducción entre los tipos de enumeración utiliza los valores enteros subyacentes:

```
HorizontalAlignment h = (HorizontalAlignment) BorderSide.Right;
// same as:
HorizontalAlignment h = (HorizontalAlignment) (int) BorderSide.Right;
```

El literal numérico 0 se trata de manera especial por el compilador en una expresión de enumeración y no requiere una conversión explícita:

```
BorderSide b = 0;    // No cast required
if (b == 0) ...
```

Existen dos razones para el tratamiento especial de 0:

- El primer miembro de una enumeración a menudo se utiliza como valor "predeterminado".
- Para tipos de enumeración combinados, 0 significa "ninguna bandera" (no flags).

Enumeraciones de banderas (Flags Enums)

Puedes combinar miembros de una enumeración. Para evitar ambigüedades, los miembros de una enumeración combinable requieren valores asignados explícitamente, generalmente en potencias de dos:

```
[Flags]
enum BorderSides { None=0, Left=1, Right=2, Top=4, Bottom=8 }
```

o:

```
enum BorderSides { None=0, Left=1, Right=1<<1, Top=1<<2, Bottom=1<<3 }
```

Para trabajar con valores combinados de la enumeración, se utilizan operadores bit a bit como | (OR) y & (AND). Estos operadores operan en los valores enteros subyacentes:

```
BorderSides leftRight = BorderSides.Left | BorderSides.Right;

if ((leftRight & BorderSides.Left) != 0)
  Console.WriteLine ("Includes Left");     // Includes Left

string formatted = leftRight.ToString();   // "Left, Right"

BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine (s == leftRight);   // True

s ^= BorderSides.Right;               // Toggles BorderSides.Right
Console.WriteLine (s);                // Left
```

Por convención, el atributo Flags siempre debe aplicarse a un tipo de enumeración cuando sus miembros son combinables. Si declaras una enumeración de este tipo sin el atributo Flags, aún puedes combinar miembros, pero llamar a ToString en una instancia de la enumeración emitirá un número en lugar de una serie de nombres.

Por convención, se da un nombre en plural a un tipo de enumeración combinable en lugar de singular. Para mayor conveniencia, puedes incluir miembros de combinación dentro de la propia declaración de la enumeración:

```
[Flags]
enum BorderSides
{
  None=0,
  Left=1, Right=1<<1, Top=1<<2, Bottom=1<<3,
  LeftRight = Left | Right,
  TopBottom = Top  | Bottom,
  All       = LeftRight | TopBottom
}
```

Operadores de Enumeración

Los operadores que funcionan con enumeraciones son:

```
=   ==   !=   <   >   <=   >=   +   -   ^  &  |   ~
+=   -=   ++  --   sizeof
```

Los operadores bit a bit, aritméticos y de comparación devuelven el resultado del procesamiento de los valores enteros subyacentes. La adición es permitida entre una enumeración y un tipo integral, pero no entre dos enumeraciones.

Problemas de seguridad de tipo

Considera la siguiente enumeración:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Debido a que una enumeración puede ser convertida hacia y desde su tipo integral subyacente, el valor real que puede tener podría estar fuera de los límites de un miembro de enumeración válido:

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine (b);                // 12345
```

Los operadores bit a bit y aritméticos pueden producir valores inválidos de manera similar:

```
BorderSide b = BorderSide.Bottom;
b++;                                  // No errors
```

Un BorderSide inválido podría romper el siguiente código:

```
void Draw (BorderSide side)
{
  if      (side == BorderSide.Left)  {...}
  else if (side == BorderSide.Right) {...}
  else if (side == BorderSide.Top)   {...}
  else                               {...} // Assume BorderSide.Bottom
}
```

Una solución es agregar otra cláusula else:

```
...
else if (side == BorderSide.Bottom) ...
else throw new ArgumentException ("Invalid BorderSide: " + side, "side");
```

Otra solución es verificar explícitamente si un valor de la enumeración es válido. El método estático Enum.IsDefined realiza esta tarea:

```
BorderSide side = (BorderSide) 12345;
Console.WriteLine (Enum.IsDefined (typeof (BorderSide), side));   // False
```

Desafortunadamente, Enum.IsDefined no funciona para enumeraciones con flags. Sin embargo, el siguiente método auxiliar (un truco dependiente del comportamiento de Enum.ToString()) devuelve true si una enumeración con flags dada es válida:

```
for (int i = 0; i <= 16; i++)
{
  BorderSides side = (BorderSides)i;
  Console.WriteLine (IsFlagDefined (side) + " " + side);
}

bool IsFlagDefined (Enum e)
{
  decimal d;
  return !decimal.TryParse(e.ToString(), out d);

}

[Flags]
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
```

Tipos anidados

Un tipo anidado se declara dentro del ámbito de otro tipo:

```
public class TopLevel
{
  public class Nested { }               // Nested class
  public enum Color { Red, Blue, Tan }  // Nested enum
}
```

Un tipo anidado tiene las siguientes características:

• Puede acceder a los miembros privados del tipo que lo contiene y a todo lo que el tipo que lo contiene puede acceder.

• Se puede declarar con la gama completa de modificadores de acceso en lugar de solo public e internal.

• La accesibilidad predeterminada para un tipo anidado es private en lugar de internal.

• Para acceder a un tipo anidado desde fuera del tipo que lo contiene, se requiere calificarlo con el nombre del tipo que lo contiene (similar a cuando se accede a miembros estáticos).

Por ejemplo, para acceder a Color.Red desde fuera de nuestra clase TopLevel, tendríamos que hacer lo siguiente:

```
TopLevel.Color color = TopLevel.Color.Red;
```

Todos los tipos (clases, estructuras, interfaces, delegados y enumeraciones) pueden ser anidados dentro de una clase o una estructura.

Aquí tienes un ejemplo de cómo acceder a un miembro privado de un tipo desde un tipo anidado:

```
public class TopLevel
{
  static int x;
  class Nested
  {
    static void Foo() { Console.WriteLine (TopLevel.x); }
  }
}
```

Aquí tienes un ejemplo de cómo aplicar el modificador de acceso protected a un tipo anidado:

```
public class TopLevel
{
  protected class Nested { }
}

public class SubTopLevel : TopLevel
{

  static void Foo() { new TopLevel.Nested(); }
}
```

Aquí tienes un ejemplo de cómo referirse a un tipo anidado desde fuera del tipo que lo contiene:

```
public class TopLevel
{
  public class Nested { }
}

class Test
{
  TopLevel.Nested n;
}
```

Los tipos anidados son utilizados ampliamente por el compilador mismo cuando genera clases privadas que capturan el estado para constructos como iteradores y métodos anónimos.



Si la única razón para usar un tipo anidado es evitar sobrecargar un espacio de nombres con demasiados tipos, considera usar un espacio de nombres anidado en su lugar. Un tipo anidado debe utilizarse debido a sus restricciones de control de acceso más fuertes, o cuando la clase anidada debe acceder a miembros privados de la clase que la contiene.

Genericos

C# tiene dos mecanismos separados para escribir código que se puede reutilizar en diferentes tipos: la herencia y los genéricos. Mientras que la herencia expresa la reutilización con un tipo base, los genéricos expresan la reutilización con una "plantilla" que contiene tipos "marcadores de posición". Los genéricos, en comparación con la herencia, pueden aumentar la seguridad de tipos y reducir la necesidad de conversiones y boxing.



Los genéricos de C# y las plantillas de C++ son conceptos similares, pero funcionan de manera diferente. Explicamos esta diferencia en "C# Generics Versus C++ Templates" en la página 157.

Tipos genéricos

Un tipo genérico declara parámetros de tipo, que son tipos de marcador de posición que serán llenados por el consumidor del tipo genérico, quien proporcionará los argumentos de tipo. Aquí tenemos un tipo genérico Stack<T>, diseñado para apilar instancias del tipo T. Stack<T> declara un único parámetro de tipo T:

```
public class Stack<T>
{
  int position;
  T[] data = new T[100];
  public void Push (T obj)  => data[position++] = obj;
  public T Pop()            => data[--position];
}
```

Podemos utilizar Stack<T> de la siguiente manera:

```
var stack = new Stack<int>();
stack.Push (5);
stack.Push (10);
int x = stack.Pop();        // x is 10
int y = stack.Pop();        // y is 5
```

Stack<int> llena el parámetro de tipo T con el argumento de tipo int, creando implícitamente un tipo sobre la marcha (la síntesis ocurre en tiempo de ejecución). Sin embargo, intentar agregar una cadena a nuestra Stack<int> produciría un error en tiempo de compilación. Stack<int> tiene efectivamente la siguiente definición (las sustituciones aparecen en negrita, con el nombre de la clase marcado para evitar confusiones):

```
public class ###
{
  int position;
  int[] data = new int[100];
  public void Push (int obj)  => data[position++] = obj;
  public int Pop()            => data[--position];
}
```

Técnicamente, decimos que Stack<T> es un tipo abierto (open type), mientras que Stack<int> es un tipo cerrado (closed type). En tiempo de ejecución, todas las instancias de tipos genéricos están cerradas, es decir, los tipos de marcador de posición están llenos. Esto significa que la siguiente declaración es ilegal:

```
var stack = new Stack<T>();   // Illegal: What is T?
```

Sin embargo, es legal si se encuentra dentro de una clase o método que define T como un parámetro de tipo:

```
public class Stack<T>
{
  ...
  public Stack<T> Clone()
  {
    Stack<T> clone = new Stack<T>();   // Legal
    ...
  }
}
```

Por qué existen los genéricos

Los genéricos existen para escribir código que sea reutilizable para diferentes tipos. Supongamos que necesitamos una pila de enteros, pero no tenemos tipos genéricos. Una solución sería codificar una versión separada de la clase para cada tipo de elemento requerido (por ejemplo, IntStack, StringStack, etc.). Claramente, esto causaría una considerable duplicación de código. Otra solución sería escribir una pila que esté generalizada utilizando object como tipo de elemento:

```
public class ObjectStack
{
  int position;
  object[] data = new object[10];


  public void Push (object obj) => data[position++] = obj;
  public object Pop()           => data[--position];
}
```

Sin embargo, una ObjectStack no funcionaría tan bien como un IntStack codificado específicamente para almacenar enteros. Un ObjectStack requeriría boxing y downcasting que no se pueden verificar en tiempo de compilación:

```
// Suppose we just want to store integers here:
ObjectStack stack = new ObjectStack();

stack.Push ("s");          // Wrong type, but no error!
int i = (int)stack.Pop();  // Downcast - runtime error
```

Lo que necesitamos es una implementación general de una pila que funcione para todos los tipos de elementos, así como una forma de especializar fácilmente esa pila para un tipo de elemento específico para aumentar la seguridad de tipos y reducir el casting y boxing. Los genéricos nos brindan precisamente esto al permitirnos parametrizar el tipo de elemento. Stack<T> tiene los beneficios tanto de ObjectStack como de IntStack. Al igual que ObjectStack, Stack<T> se escribe una vez para funcionar de manera general para todos los tipos. Al igual que IntStack, Stack<T> se especializa para un tipo particular, y la belleza es que este tipo es T, que sustituimos sobre la marcha.



ObjectStack es equivalente en funcionalidad a Stack<object>.

Metodos genericos

Los métodos genéricos permiten declarar parámetros de tipo dentro de la firma de un método. Con los métodos genéricos, se pueden implementar muchos algoritmos fundamentales de una manera general. Aquí hay un ejemplo de un método genérico que intercambia el contenido de dos variables de cualquier tipo T:

```
static void Swap<T> (ref T a, ref T b)
{
  T temp = a;
  a = b;
  b = temp;
}
```

El método Swap<T> se llama de la siguiente manera:

```
int x = 5;
int y = 10;
Swap (ref x, ref y);
```

Generalmente, no es necesario proporcionar argumentos de tipo a un método genérico, porque el compilador puede inferir implícitamente el tipo. Si hay ambigüedad, los métodos genéricos pueden ser llamados con argumentos de tipo de la siguiente manera:

```
Swap<int> (ref x, ref y);
```

Dentro de un tipo genérico, un método no se considera genérico a menos que introduzca parámetros de tipo (con la sintaxis de corchetes angulares). El método Pop en nuestra pila genérica simplemente utiliza el parámetro de tipo existente, T, y no se considera un método genérico.

Los métodos y los tipos son las únicas construcciones que pueden introducir parámetros de tipo. Las propiedades, indexadores, eventos, campos, constructores, operadores, etc., no pueden declarar parámetros de tipo, aunque pueden participar en los parámetros de tipo ya declarados por su tipo contenedor. En nuestro ejemplo de pila genérica, por ejemplo, podríamos escribir un indexador que devuelve un elemento genérico:

```
public T this [int index] => data [index];
```

De manera similar, los constructores pueden participar en los parámetros de tipo existentes, pero no introducirlos:

```
public Stack<T>() { }   // Illegal
```

Declaración de parámetros de tipo

Los parámetros de tipo se pueden introducir en la declaración de clases, estructuras, interfaces, delegados (tratados en el Capítulo 4) y métodos. Otros constructos, como propiedades, no pueden introducir un parámetro de tipo, pero pueden usar uno. Por ejemplo, la propiedad Value utiliza T:

```
public struct Nullable<T>
{
  public T Value { get; }
}
```

Un tipo o método genérico puede tener varios parámetros:

```
class Dictionary<TKey, TValue> {...}
```

Para instanciar:

```
Dictionary<int,string> myDict = new Dictionary<int,string>();
```

O también:

```
var myDict = new Dictionary<int,string>();
```

Los nombres de tipos genéricos y los nombres de métodos pueden tener sobrecarga siempre que el número de parámetros de tipo sea diferente. Por ejemplo, los siguientes tres nombres de tipos no entran en conflicto:

```
class A        {}
class A<T>     {}
class A<T1,T2> {}
```



Por convención, los tipos genéricos y los métodos con un solo parámetro de tipo generalmente nombran su parámetro como T, siempre y cuando la intención del parámetro sea clara. Cuando se utilizan múltiples parámetros de tipo, cada parámetro se antepone con T pero tiene un nombre más descriptivo.

typeof y tipos genéricos sin enlazar

Los tipos genéricos abiertos no existen en tiempo de ejecución: se cierran como parte de la compilación. Sin embargo, es posible que un tipo genérico sin enlazar exista en tiempo de ejecución, únicamente como un objeto de tipo Type. La única forma de especificar un tipo genérico sin enlazar en C# es a través del operador typeof:

```
class A<T> {}
class A<T1,T2> {}
...

Type a1 = typeof (A<>);   // Unbound type (notice no type arguments).
Type a2 = typeof (A<,>);  // Use commas to indicate multiple type args.
```

Los tipos genéricos abiertos se utilizan en conjunto con la API de reflexión (Capítulo 18). También puedes utilizar el operador typeof para especificar un tipo cerrado:

```
Type a3 = typeof (A<int,int>);
```

O puedes especificar un tipo abierto (que se cierra en tiempo de ejecución):

```
class B<T> { void X() { Type t = typeof (T); } }
```

El valor genérico predeterminado

Puedes utilizar la palabra clave default para obtener el valor predeterminado de un parámetro de tipo genérico. El valor predeterminado para un tipo de referencia es null, y el valor predeterminado para un tipo de valor es el resultado de establecer en cero los bits de los campos del tipo de valor:

```
static void Zap<T> (T[] array)
{
  for (int i = 0; i < array.Length; i++)
    array[i] = default(T);
}
```

A partir de C# 7.1, puedes omitir el argumento de tipo en los casos en los que el compilador pueda inferirlo. Podríamos reemplazar la última línea de código con esto:

```
array[i] = default;
```

Restricciones genéricas

Por defecto, puedes sustituir un parámetro de tipo con cualquier tipo. Se pueden aplicar restricciones a un parámetro de tipo para requerir argumentos de tipo más específicos. Estas son las posibles restricciones:

```
where T : base-class   // Base-class constraint
where T : interface    // Interface constraint
where T : class        // Reference-type constraint
where T : class?       // (see "Nullable Reference Types" in Chapter 1)
where T : struct       // Value-type constraint (excludes Nullable types)
where T : unmanaged    // Unmanaged constraint
where T : new()        // Parameterless constructor constraint
where U : T            // Naked type constraint


where T : notnull      // Non-nullable value type, or (from C# 8)
                       // a non-nullable reference type
```

En el siguiente ejemplo, GenericClass<T,U> requiere que T derive de (o sea idéntico a) SomeClass e implemente Interface1, y requiere que U proporcione un constructor sin parámetros:

```
class     SomeClass {}
interface Interface1 {}

class GenericClass<T,U> where T : SomeClass, Interface1
                        where U : new()
{...}
```

Puedes aplicar restricciones donde sea que se definan los parámetros de tipo, tanto en métodos como en definiciones de tipo.

Una restricción de clase base especifica que el parámetro de tipo debe ser una subclase (o coincidir) de una clase en particular; una restricción de interfaz especifica que el parámetro de

tipo debe implementar esa interfaz. Estas restricciones permiten que las instancias del parámetro de tipo se conviertan implícitamente en esa clase o interfaz. Por ejemplo, supongamos que queremos escribir un método genérico Max, que devuelve el máximo de dos valores. Podemos aprovechar la interfaz genérica definida en el espacio de nombres System llamada IComparable<T>:

```
public interface IComparable<T>   // Simplified version of interface
{
  int CompareTo (T other);
}
```

CompareTo devuelve un número positivo si "this" es mayor que "other". Usando esta interfaz como restricción, podemos escribir un método Max de la siguiente manera (omitimos la comprobación de nulos por simplicidad):

```
static T Max <T> (T a, T b) where T : IComparable<T>
{
  return a.CompareTo (b) > 0 ? a : b;
}
```

El método Max puede aceptar argumentos de cualquier tipo que implemente IComparable<T> (lo cual incluye la mayoría de los tipos incorporados como int y string):

```
int z = Max (5, 10);               // 10
string last = Max ("ant", "zoo");  // zoo
```

La restricción de clase y la restricción de struct especifican que T debe ser un tipo de referencia o un tipo de valor (no nullable). Un excelente ejemplo de la restricción de struct es la estructura System.Nullable<T> (la cual se discute en detalle en "Tipos de Valor Nullable" en la página 194).

```
struct Nullable<T> where T : struct {...}
```

La restricción "unmanaged" (introducida en C# 7.3) es una versión más estricta de la restricción "struct": T debe ser un tipo de valor simple o una estructura que esté (de forma recursiva) libre de cualquier tipo de referencia.

La restricción de constructor sin parámetros requiere que T tenga un constructor público sin parámetros. Si se define esta restricción, puedes llamar a "new()" en T:

```
static void Initialize<T> (T[] array) where T : new()
{
  for (int i = 0; i < array.Length; i++)
    array[i] = new T();
}
```

La restricción de tipo desnudo requiere que un parámetro de tipo derive de (o coincida con) otro parámetro de tipo. En este ejemplo, el método "FilteredStack" devuelve otro Stack que contiene solo el subconjunto de elementos donde el parámetro de tipo U es del parámetro de tipo T:

```
class Stack<T>
{
  Stack<U> FilteredStack<U>() where U : T {...}
}
```

Subclases de tipos genéricos

La herencia de clases genéricas funciona de la misma manera que con las clases no genéricas. La subclase puede dejar abiertos los parámetros de tipo de la clase base, como en el siguiente ejemplo:

```
class Stack<T>                   {...}
class SpecialStack<T> : Stack<T> {...}
```

O la subclase puede cerrar los parámetros de tipo genérico con un tipo concreto:

```
class IntStack : Stack<int>  {...}
```

Un subtipo también puede introducir nuevos argumentos de tipo:

```
class List<T>                     {...}
class KeyedList<T,TKey> : List<T> {...}
```



Técnicamente, todos los argumentos de tipo en un subtipo son nuevos: podríamos decir que un subtipo cierra y luego vuelve a abrir los argumentos de tipo base. Esto significa que una subclase puede dar nombres nuevos (y potencialmente más significativos) a los argumentos de tipo que vuelve a abrir:

```
class List<T> {...}
class KeyedList<TElement,TKey> : List<TElement> {...}
```

Declaraciones genéricas auto-referenciadas

Un tipo puede nombrarse a sí mismo como el tipo concreto al cerrar un argumento de tipo:

```
public interface IEquatable<T> { bool Equals (T obj); }

public class Balloon : IEquatable<Balloon>


  {
    public string Color { get; set; }
    public int CC { get; set; }

    public bool Equals (Balloon b)
    {
      if (b == null) return false;
      return b.Color == Color && b.CC == CC;
    }
  }
```

También son válidos los siguientes ejemplos:

```
class Foo<T> where T : IComparable<T> { ... }
class Bar<T> where T : Bar<T> { ... }
```

Datos estáticos

Los datos estáticos son únicos para cada tipo cerrado:

```
Console.WriteLine (++Bob<int>.Count);     // 1
Console.WriteLine (++Bob<int>.Count);     // 2
Console.WriteLine (++Bob<string>.Count);  // 1
Console.WriteLine (++Bob<object>.Count);  // 1

class Bob<T> { public static int Count; }
```

Parámetros de tipo y conversiones

El operador de conversión en C# puede realizar varios tipos de conversiones, incluyendo:

- Conversión numérica
- Conversión de referencia
- Conversión de boxeo/desboxeo
- Conversión personalizada (a través de la sobrecarga de operadores; ver Capítulo 4)

La decisión sobre qué tipo de conversión se realizará se toma en tiempo de compilación, basándose en los tipos conocidos de los operandos. Esto crea un escenario interesante con los parámetros de tipo genérico, ya que los tipos precisos de los operandos son desconocidos en tiempo de compilación. Si esto conduce a ambigüedad, el compilador genera un error.

El escenario más común es cuando se desea realizar una conversión de referencia:

```
StringBuilder Foo<T> (T arg)
{
  if (arg is StringBuilder)
    return (StringBuilder) arg;   // Will not compile
  ...
}
```

Sin conocer el tipo real de T, el compilador se preocupa de que puedas haber pretendido que esto sea una conversión personalizada. La solución más sencilla es utilizar el operador as, que no tiene ambigüedad ya que no puede realizar conversiones personalizadas:

```
StringBuilder Foo<T> (T arg)
{
  StringBuilder sb = arg as StringBuilder;
  if (sb != null) return sb;
  ...
}
```

Una solución más general es realizar primero una conversión a objeto. Esto funciona porque las conversiones hacia/desde objeto se asumen como conversiones de referencia o de boxing/unboxing. En este caso, StringBuilder es un tipo de referencia, por lo que debe ser una conversión de referencia:

```
return (StringBuilder) (object) arg;
```

Las conversiones de unboxing también pueden presentar ambigüedades. Lo siguiente podría ser una conversión de unboxing, numérica o personalizada:

```
int Foo<T> (T x) => (int) x;      // Compile-time error
```

La solución, nuevamente, es realizar primero una conversión a objeto y luego a int (lo cual indica de manera inequívoca una conversión de unboxing en este caso):

```
int Foo<T> (T x) => (int) (object) x;
```

Covariancia

Suponiendo que A es convertible a B, X tiene un parámetro de tipo covariante si X<A> es convertible a X<B>.



Con la noción de covarianza (y contravarianza) en C#, "convertible" significa convertible a través de una conversión implícita de referencia, como cuando A es una subclase de B o A implementa B. Las conversiones numéricas, las conversiones de boxing y las conversiones personalizadas no están incluidas en esta definición.

Por ejemplo, el tipo IFoo<T> tiene un parámetro T covariante si lo siguiente es legal:

```
IFoo<string> s = ...;
IFoo<object> b = s;
```

Las interfaces permiten parámetros de tipo covariantes (al igual que los delegados; ver Capítulo 4), pero las clases no lo permiten. Los arrays también permiten la covarianza (A[] puede convertirse a B[] si A tiene una conversión implícita de referencia a B) y se discuten aquí para hacer una comparación.



La covarianza y la contravarianza (o simplemente "varianza") son conceptos avanzados. La motivación detrás de la introducción y mejora de la varianza en C# fue permitir que las interfaces genéricas y los tipos genéricos (en particular, aquellos definidos en .NET, como IEnumerable<T>) funcionen de manera más intuitiva. Puedes beneficiarte de esto sin necesidad de comprender los detalles de la covarianza y la contravarianza.

La varianza no es automática

Para garantizar la seguridad de tipos estática, los parámetros de tipo no son automáticamente variantes. Considera el siguiente ejemplo:

```
class Animal {}
class Bear : Animal {}
class Camel : Animal {}

public class Stack<T>   // A simple Stack implementation
{
  int position;
  T[] data = new T[100];
  public void Push (T obj)  => data[position++] = obj;
  public T Pop()            => data[--position];
}
```

El siguiente código no compilará:

```
Stack<Bear> bears = new Stack<Bear>();
Stack<Animal> animals = bears;            // Compile-time error
```

Esta restricción evita la posibilidad de fallas en tiempo de ejecución con el siguiente código:

```
animals.Push (new Camel());      // Trying to add Camel to bears
```

Sin embargo, la falta de covarianza puede obstaculizar la reutilización. Por ejemplo, supongamos que queremos escribir un método para lavar una pila de animales:

```
  public class ZooCleaner
  {
    public static void Wash (Stack<Animal> animals) {...}
  }
```

Llamar a `Wash` con una pila de osos generaría un error en tiempo de compilación. Una solución es redefinir el método `Wash` con una restricción:

```
class ZooCleaner
{
  public static void Wash<T> (Stack<T> animals) where T : Animal { ... }
}
```

Ahora podemos llamar a `Wash` de la siguiente manera:

```
Stack<Bear> bears = new Stack<Bear>();
ZooCleaner.Wash (bears);
```

Otra solución es hacer que `Stack<T>` implemente una interfaz con un parámetro de tipo covariante, como veremos más adelante.

Arrays

Por razones históricas, los tipos de array admiten covarianza. Esto significa que `B[]` se puede convertir a `A[]` si `B` es una subclase de `A` (y ambos son tipos de referencia):

```
Bear[] bears = new Bear[3];
Animal[] animals = bears;     // OK
```

El inconveniente de esta reutilización es que las asignaciones de elementos pueden fallar en tiempo de ejecución:

```
animals[0] = new Camel();      // Runtime error
```

Declarar un parámetro de tipo covariante

Los parámetros de tipo en interfaces y delegados se pueden declarar como covariantes al marcarlos con el modificador `out`. Este modificador garantiza que, a diferencia de los arrays, los parámetros de tipo covariantes son completamente seguros en cuanto a tipos.

Podemos ilustrar esto con nuestra clase `Stack<T>` al hacerla implementar la siguiente interfaz:

```
public interface IPoppable<out T> { T Pop(); }
```

El modificador `out` en `T` indica que `T` se usa solo en posiciones de salida (por ejemplo, tipos de retorno para métodos). El modificador `out` marca el parámetro de tipo como covariante y nos permite hacer lo siguiente:

```
var bears = new Stack<Bear>();
bears.Push (new Bear());
// Bears implements IPoppable<Bear>. We can convert to IPoppable<Animal>:
IPoppable<Animal> animals = bears;   // Legal
Animal a = animals.Pop();
```

La conversión de `bears` a `animals` es permitida por el compilador, gracias al parámetro de tipo siendo covariante. Esto es seguro en cuanto a tipos porque el caso que el compilador está tratando de evitar, como empujar un `Camel` en la pila, no puede ocurrir, ya que no hay forma de pasar un `Camel` a una interfaz donde `T` solo puede aparecer en posiciones de salida.


La covariancia (y contravariancia) en las interfaces es algo que normalmente se consume: es menos común que necesites escribir interfaces variantes.


Curiosamente, los parámetros de método marcados como "out" no son elegibles para covariancia debido a una limitación en el CLR (Common Language Runtime).

Podemos aprovechar la capacidad de realizar conversiones covariantes para resolver el problema de reutilización descrito anteriormente.

```
public class ZooCleaner
{
  public static void Wash (IPoppable<Animal> animals) { ... }
}
```



Las interfaces IEnumerator<T> e IEnumerable<T> descritas en el Capítulo 7 tienen un parámetro T covariante. Esto permite que se pueda realizar una conversión de IEnumerable<string> a IEnumerable<object>, por ejemplo.

El compilador generará un error si se utiliza un parámetro de tipo covariante en una posición de entrada (por ejemplo, un parámetro de un método o una propiedad de escritura).



La covariancia (y contravariancia) solo funciona para elementos con conversiones de referencia, no conversiones de boxing. Esto se aplica tanto a la covariancia de parámetros de tipo como a la covariancia de arrays. Por lo tanto, si escribieras un método que acepta un parámetro de tipo `IPoppable<object>`, podrías llamarlo con `IPoppable<string>`, pero no con `IPoppable<int>`.

Contravariancia

Anteriormente vimos que, asumiendo que `A` permite una conversión implícita de referencia a `B`, un tipo `X` tiene un parámetro de tipo covariante si `X<A>` permite una conversión de referencia a `X<B>`. La contravariancia ocurre cuando puedes convertir en la dirección opuesta, es decir, de `X<B>` a `X<A>`. Esto se admite si el parámetro de tipo aparece solo en posiciones de entrada y se designa con el modificador `in`. Extendiendo nuestro ejemplo anterior, supongamos que la clase `Stack<T>` implementa la siguiente interfaz:

```
public interface IPushable<in T> { void Push (T obj); }
```

Ahora podemos hacer lo siguiente legalmente:

```
IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals;    // Legal
bears.Push (new Bear());
```

Ningún miembro en `IPushable` devuelve un `T`, por lo que no podemos tener problemas al convertir `animals` a `bears` (no hay forma de hacer `Pop`, por ejemplo, a través de esa interfaz).



Nuestra clase `Stack<T>` puede implementar tanto `IPushable<T>` como `IPoppable<T>`, ¡a pesar de que `T` tenga anotaciones de covarianza opuestas en las dos interfaces! Esto funciona porque debes ejercer la covarianza a través de la interfaz y no de la clase; por lo tanto, debes comprometerte con el enfoque de `IPoppable` o `IPushable` antes de realizar una conversión variante. Este enfoque luego te restringe a las operaciones que son legales según las reglas de covarianza o contravarianza correspondientes.

Esto también ilustra por qué las clases no permiten parámetros de tipo variantes: las implementaciones concretas generalmente requieren que los datos fluyan en ambas direcciones.

Para dar otro ejemplo, considera la siguiente interfaz, definida en el espacio de nombres System:

```
public interface IComparer<in T>
{
  // Returns a value indicating the relative ordering of a and b
  int Compare (T a, T b);
}
```

Debido a que la interfaz tiene un parámetro de tipo contravariante T, podemos usar un IComparer<object> para comparar dos cadenas:

```
var objectComparer = Comparer<object>.Default;
// objectComparer implements IComparer<object>
IComparer<string> stringComparer = objectComparer;
int result = stringComparer.Compare ("Brett", "Jemaine");
```

Al igual que con la covarianza, el compilador mostrará un error si intentas usar un parámetro de tipo contravariante en una posición de salida (por ejemplo, como valor de retorno o en una propiedad de solo lectura).

C# Generics Versus C++ Plantillas(Templates)

Las generics de C# y las plantillas de C++ son similares en aplicación, pero funcionan de manera muy diferente. En ambos casos, debe producirse una síntesis entre el productor y el consumidor en la que los tipos de marcador del productor son reemplazados por el consumidor. Sin embargo, con las generics de C#, los tipos de productor (es decir, los tipos abiertos como List<T>) se pueden compilar en una biblioteca (como mscorlib.dll). Esto funciona porque la síntesis entre el productor y el consumidor que produce tipos cerrados en realidad no ocurre hasta en tiempo de ejecución. Con las plantillas de C++, esta síntesis se realiza en tiempo de compilación. Esto significa que en C++ no se implementan bibliotecas de plantillas como archivos .dll, sino que existen solo como código fuente. También dificulta la inspección dinámica y la creación de tipos parametrizados sobre la marcha.

Para profundizar en el motivo de esto, consideremos nuevamente el método Max en C#:

```
static T Max <T> (T a, T b) where T : IComparable<T>
  => a.CompareTo (b) > 0 ? a : b;
```

¿Por qué no podríamos haberlo implementado de esta manera?

```
static T Max <T> (T a, T b)
  => (a > b ? a : b);            // Compile error
```

La razón es que Max necesita compilarse una vez y funcionar para todos los posibles valores de T. La compilación no puede tener éxito porque no hay un único significado para el operador > en todos los valores de T, de hecho, no todos los T tienen un operador >. En contraste, el siguiente código muestra el mismo método Max escrito con plantillas de C++. Este código se compilará por

separado para cada valor de T, adoptando las semánticas que > tiene para un T en particular y fallando en la compilación si un T particular no admite el operador >:

```
template <class T> T Max (T a, T b)
{
  return a > b ? a : b;
}
```

4 C# Avanzado

En este capítulo, cubrimos temas avanzados de C# que se basan en conceptos explorados en los Capítulos 2 y 3. Debes leer las primeras cuatro secciones en secuencia; las secciones restantes las puedes leer en cualquier orden.

Delegados

Un delegado es un objeto que sabe cómo llamar a un método.

Un tipo de delegado define el tipo de método que las instancias de delegado pueden llamar. Específicamente, define el tipo de retorno del método y los tipos de sus parámetros. A continuación se define un tipo de delegado llamado "Transformer":

```
delegate int Transformer (int x);
```

El delegado "Transformer" es compatible con cualquier método que tenga un tipo de retorno int y un único parámetro int, como este:

```
int Square (int x) { return x * x; }
```

O de forma más concisa:

```
int Square (int x) => x * x;
```

Asignar un método a una variable de delegado crea una instancia de delegado:

```
Transformer t = Square;
```

Puedes invocar una instancia de delegado de la misma manera que un método:

```
int answer = t(3);    // answer is 9
```

Aquí tienes un ejemplo completo:

```
Transformer t = Square;          // Create delegate instance
int result = t(3);               // Invoke delegate
Console.WriteLine (result);      // 9

int Square (int x) => x * x;


delegate int Transformer (int x);   // Delegate type declaration
```

Una instancia de delegado actúa literalmente como un delegado para el llamador: el llamador invoca al delegado y luego el delegado llama al método objetivo. Esta indirección desacopla al llamador del método objetivo.

La declaración

```
Transformer t = Square;
```

es una forma abreviada de:

```
Transformer t = new Transformer (Square);
```



Técnicamente, estamos especificando un grupo de métodos cuando nos referimos a Square sin paréntesis ni argumentos. Si el método está sobrecargado, C# elegirá la sobrecarga correcta basándose en la firma del delegado al que se está asignando.

La expresión

```
t(3)
```

es una forma abreviada de:

```
t.Invoke(3)
```



Un delegado es similar a un callback, un término general que abarca construcciones como los punteros a funciones en C.

Escribir métodos de complementos con delegados

Una variable de delegado se asigna a un método en tiempo de ejecución. Esto es útil para escribir métodos de complementos. En este ejemplo, tenemos un método de utilidad llamado Transform que aplica una transformación a cada elemento en una matriz de enteros. El método Transform tiene un parámetro de tipo delegado, que se puede utilizar para especificar una transformación de complemento:

```
int[] values = { 1, 2, 3 };
Transform (values, Square);      // Hook in the Square method

foreach (int i in values)
  Console.Write (i + "  ");      // 1   4   9

void Transform (int[] values, Transformer t)
{
  for (int i = 0; i < values.Length; i++)
    values[i] = t (values[i]);
}

int Square (int x) => x * x;
int Cube (int x) => x * x * x;

delegate int Transformer (int x);
```

Podemos cambiar la transformación simplemente cambiando Square por Cube en la segunda línea de código.

Nuestro método Transform es una función de orden superior porque es una función que toma una función como argumento. (Un método que devuelve un delegado también sería una función de orden superior).

Destinos de método de instancia y estáticos

El método de destino de un delegado puede ser un método local, estático o de instancia. Lo siguiente ilustra un método de destino estático:

```
Transformer t = Test.Square;
Console.WriteLine (t(10));      // 100

class Test { public static int Square (int x) => x * x; }

delegate int Transformer (int x);
```

Lo siguiente ilustra un método de destino de instancia:

```
Test test = new Test();
Transformer t = test.Square;
Console.WriteLine (t(10));      // 100

class Test { public int Square (int x) => x * x; }

delegate int Transformer (int x);
```

Cuando se asigna un método de instancia a un objeto delegado, este último mantiene una referencia no solo al método, sino también a la instancia a la que pertenece el método. La propiedad Target de la clase System.Delegate representa esta instancia (y será nula para un delegado que haga referencia a un método estático). Aquí tienes un ejemplo:

```
MyReporter r = new MyReporter();
r.Prefix = "%Complete: ";
ProgressReporter p = r.ReportProgress;
p(99);                                  // %Complete: 99
Console.WriteLine (p.Target == r);      // True
Console.WriteLine (p.Method);           // Void ReportProgress(Int32)
r.Prefix = "";
p(99);                                  // 99

public delegate void ProgressReporter (int percentComplete);

class MyReporter
{
  public string Prefix = "";

  public void ReportProgress (int percentComplete)
    => Console.WriteLine (Prefix + percentComplete);
}
```

Debido a que la instancia se almacena en la propiedad Target del delegado, su tiempo de vida se extiende al menos hasta el tiempo de vida del delegado.

Delegados Multicast

Todos los delegados tienen la capacidad de ser multicast. Esto significa que una instancia de delegado puede hacer referencia no solo a un único método de destino, sino también a una lista de métodos de destino. Los operadores + y += combinan las instancias de delegado:

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

La última línea es funcionalmente equivalente a lo siguiente:

```
d = d + SomeMethod2;
```

Al invocar d, se llamarán tanto SomeMethod1 como SomeMethod2. Los delegados se invocan en el orden en que se agregan.

Los operadores - y -= eliminan el delegado derecho del delegado izquierdo:

```
d -= SomeMethod1;
```

Al invocar d ahora solo se invocará SomeMethod2.

Llamar a + o += en una variable de delegado con un valor nulo funciona y es equivalente a asignarle un nuevo valor:

```
SomeDelegate d = null;
d += SomeMethod1;       // Equivalent (when d is null) to d = SomeMethod1;
```

De manera similar, llamar a -= en una variable de delegado con un único destino coincidente es equivalente a asignarle null a esa variable.



Los delegados son inmutables, por lo que cuando llamas a += o -=, en realidad estás creando una nueva instancia de delegado y asignándola a la variable existente.

Si un delegado multicast tiene un tipo de retorno distinto de void, el llamador recibirá el valor de retorno del último método invocado. Los métodos anteriores aún se llaman, pero sus valores de retorno se descartan. En la mayoría de los escenarios en los que se utilizan delegados multicast, tienen un tipo de retorno void, por lo que esta sutileza no surge.



Todos los tipos de delegado derivan implícitamente de System.MulticastDelegate, que a su vez hereda de System.Delegate. C# compila las operaciones +, -, += y -= realizadas en un delegado en los métodos estáticos Combine y Remove de la clase System.Delegate.

Ejemplo de un delegado multicast

Supongamos que has escrito un método que tarda mucho tiempo en ejecutarse. Ese método podría informar regularmente sobre el progreso a quien lo llama invocando un delegado. En este

ejemplo, el método HardWork tiene un parámetro de tipo ProgressReporter, que invoca para indicar el progreso:

```
public delegate void ProgressReporter (int percentComplete);

public class Util
{
  public static void HardWork (ProgressReporter p)
  {
    for (int i = 0; i < 10; i++)
    {
      p (i * 10);                          // Invoke delegate
      System.Threading.Thread.Sleep (100);  // Simulate hard work
    }
  }
}
```

Para monitorizar el progreso, podemos crear una instancia de delegado multicast `p`, de modo que el progreso se supervisa mediante dos métodos independientes:

```
ProgressReporter p = WriteProgressToConsole;
p += WriteProgressToFile;
Util.HardWork (p);

void WriteProgressToConsole (int percentComplete)
  => Console.WriteLine (percentComplete);

void WriteProgressToFile (int percentComplete)
  => System.IO.File.WriteAllText ("progress.txt",
                                  percentComplete.ToString());
```

Tipos de Delegados Genéricos

Un tipo de delegado puede contener parámetros de tipo genérico:

```
public delegate T Transformer<T> (T arg);
```

Con esta definición, podemos escribir un método de utilidad generalizado `Transform` que funcione con cualquier tipo:

```
int[] values = { 1, 2, 3 };
Util.Transform (values, Square);      // Hook in Square
foreach (int i in values)
  Console.Write (i + "  ");           // 1   4   9

int Square (int x) => x * x;

public class Util
{
  public static void Transform<T> (T[] values, Transformer<T> t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t (values[i]);
  }
}
```

Los delegados Func y Action

Con los delegados genéricos, es posible escribir un conjunto pequeño de tipos de delegados que son tan generales que pueden funcionar para métodos de cualquier tipo de retorno y cualquier número de argumentos (razonable). Estos delegados son los delegados Func y Action, definidos en el espacio de nombres System (las anotaciones in y out indican covarianza y contravarianza, que cubriremos en el contexto de los delegados próximamente):

```
delegate TResult Func <out TResult>                ();
delegate TResult Func <in T, out TResult>          (T arg);
delegate TResult Func <in T1, in T2, out TResult>  (T1 arg1, T2 arg2);
... and so on, up to T16

delegate void Action                 ();
delegate void Action <in T>          (T arg);
delegate void Action <in T1, in T2>  (T1 arg1, T2 arg2);
... and so on, up to T16
```

Estos delegados son extremadamente generales. El delegado Transformer en nuestro ejemplo anterior puede ser reemplazado por un delegado Func que toma un argumento de tipo T y devuelve un valor del mismo tipo:

```
public static void Transform<T> (T[] values, Func<T,T> transformer)
{
  for (int i = 0; i < values.Length; i++)
    values[i] = transformer (values[i]);
}
```

Los únicos escenarios prácticos que no están cubiertos por estos delegados son los parámetros ref/out y los punteros.



Cuando C# fue introducido por primera vez, los delegados Func y Action no existían (porque no existían los genéricos). Es por esta razón histórica que gran parte de .NET utiliza tipos de delegados personalizados en lugar de Func y Action.

Delegados vs Interfaces

Un problema que se puede resolver con un delegado también se puede resolver con una interfaz. Por ejemplo, podemos reescribir nuestro ejemplo original con una interfaz llamada ITransformer en lugar de un delegado:

```
int[] values = { 1, 2, 3 };
Util.TransformAll (values, new Squarer());
foreach (int i in values)
  Console.WriteLine (i);

public interface ITransformer
{
  int Transform (int x);
}

public class Util
{
```

```
 public static void TransformAll (int[] values, ITransformer t)
 {
   for (int i = 0; i < values.Length; i++)
     values[i] = t.Transform (values[i]);
 }
}

class Squarer : ITransformer
{
  public int Transform (int x) => x * x;
}
```

Un diseño de delegado podría ser una mejor opción que un diseño de interfaz si se cumplen una o más de estas condiciones:

• La interfaz solo define un método.

• Se necesita capacidad de multicast.

• El suscriptor necesita implementar la interfaz varias veces.

En el ejemplo de ITransformer, no necesitamos multicast. Sin embargo, la interfaz solo define un método. Además, es posible que nuestro suscriptor necesite implementar ITransformer varias veces para admitir transformaciones diferentes, como cuadrado o cubo. Con las interfaces, nos vemos obligados a escribir un tipo separado por cada transformación, ya que una clase solo puede implementar ITransformer una vez. Esto resulta bastante engorroso:

```
int[] values = { 1, 2, 3 };
Util.TransformAll (values, new Cuber());
foreach (int i in values)
  Console.WriteLine (i);

class Squarer : ITransformer
{
  public int Transform (int x) => x * x;
}

class Cuber : ITransformer
{
  public int Transform (int x) => x * x * x;
}
```

Compatibilidad de delegados

Compatibilidad de tipos

Los tipos de delegado son todos incompatibles entre sí, incluso si sus firmas son iguales:

```
D1 d1 = Method1;
D2 d2 = d1;                          // Compile-time error

void Method1() { }
```

```
delegate void D1();
delegate void D2();
```



Lo siguiente, sin embargo, está permitido:

```
D2 d2 = new D2 (d1);
```

Las instancias de delegados se consideran iguales si tienen los mismos objetivos de método:

```
D d1 = Method1;
D d2 = Method1;
Console.WriteLine (d1 == d2);         // True

void Method1() { }
delegate void D();
```

Los delegados multicast se consideran iguales si hacen referencia a los mismos métodos en el mismo orden.

Compatibilidad de parámetros

Cuando llamas a un método, puedes proporcionar argumentos que tienen tipos más específicos que los parámetros de ese método. Esto es un comportamiento polimórfico común. Por la misma razón, un delegado puede tener tipos de parámetros más específicos que su objetivo de método. Esto se llama contravarianza. Aquí tienes un ejemplo:

```
StringAction sa = new StringAction (ActOnObject);
sa ("hello");

void ActOnObject (object o) => Console.WriteLine (o);   // hello

delegate void StringAction (string s);
```

(Al igual que con la variación del parámetro de tipo, los delegados son variantes solo para conversiones de referencia).

Un delegado simplemente llama a un método en nombre de otra persona. En este caso, se invoca la acción de cadena con un argumento de tipo cadena. Cuando el argumento se transmite al método objetivo, el argumento se convierte implícitamente a objeto.



El patrón de eventos estándar está diseñado para ayudarte a utilizar la contravarianza a través de su uso de la clase base común EventArgs. Por ejemplo, puedes tener un único método invocado por dos delegados diferentes, uno pasando un MouseEventArgs y el otro pasando un KeyEventArgs.

Compatibilidad de tipos de retorno

Si llamas a un método, es posible que obtengas un tipo más específico de lo que pediste. Esto es un comportamiento polimórfico común. Por la misma razón, el método objetivo de un delegado puede devolver un tipo más específico que el descrito por el delegado. Esto se llama covarianza:

```
ObjectRetriever o = new ObjectRetriever (RetriveString);
object result = o();
Console.WriteLine (result);      // hello

string RetriveString() => "hello";

delegate object ObjectRetriever();
```

ObjectRetriever espera obtener un objeto, pero también puede aceptar una subclase de objeto. Los tipos de retorno de los delegados son covariantes.

Variance de parámetro de tipo de delegado genérico

En el Capítulo 3, vimos cómo las interfaces genéricas admiten parámetros de tipo covariantes y contravariantes. La misma capacidad existe también para los delegados.

Si estás definiendo un tipo de delegado genérico, es buena práctica hacer lo siguiente:

- Marcar un parámetro de tipo utilizado solo en el valor de retorno como covariante (out).
- Marcar cualquier parámetro de tipo utilizado solo en los parámetros como contravariante (in).

Hacer esto permite que las conversiones funcionen de manera natural, respetando las relaciones de herencia entre los tipos.

El siguiente delegado (definido en el espacio de nombres System) tiene un TResult covariante:

```
delegate TResult Func<out TResult>();
```

Esto permite:

```
Func<string> x = ...;
Func<object> y = x;
```

El siguiente delegado (definido en el espacio de nombres System) tiene un T contravariante:

```
delegate void Action<in T> (T arg);
```

Esto permite:

```
Action<object> x = ...;
Action<string> y = x;
```

Eventos

Al utilizar delegados, suelen aparecer dos roles emergentes: el emisor y el suscriptor.

El emisor es un tipo que contiene un campo de delegado. El emisor decide cuándo emitir eventos invocando el delegado.

Los suscriptores son los receptores del método objetivo. Un suscriptor decide cuándo comenzar y dejar de escuchar llamando a += y -= en el delegado del emisor. Un suscriptor no conoce ni interfiere con otros suscriptores.

Los eventos son una característica del lenguaje que formaliza este patrón. Un evento es una construcción que expone solo el subconjunto de características del delegado requeridas para el modelo de emisor/suscriptor. El objetivo principal de los eventos es evitar que los suscriptores interfieran entre sí.

La forma más sencilla de declarar un evento es colocar la palabra clave "event" delante de un miembro delegado:

```
// Delegate definition
public delegate void PriceChangedHandler (decimal oldPrice,
                                          decimal newPrice);
public class Broadcaster
{
  // Event declaration
  public event PriceChangedHandler PriceChanged;
}
```

El código dentro del tipo "Broadcaster" tiene acceso completo a "PriceChanged" y puede tratarlo como un delegado. El código fuera de "Broadcaster" solo puede realizar operaciones += y -= en el evento "PriceChanged".

¿Cómo funcionan los eventos en el interior?

Cuando declaras un evento de la siguiente manera:

```
public class Broadcaster
{
  public event PriceChangedHandler PriceChanged;
}
```

Suceden tres cosas internamente:

Primero, el compilador traduce la declaración del evento a algo similar a lo siguiente:

```
PriceChangedHandler priceChanged;   // private delegate
public event PriceChangedHandler PriceChanged
{
  add    { priceChanged += value; }
  remove { priceChanged -= value; }
}
```

Las palabras clave "add" y "remove" denotan los accesores de eventos explícitos, que actúan de manera similar a los accesores de propiedades. Describiremos cómo escribirlos más adelante.

Segundo, el compilador busca dentro de la clase "Broadcaster" las referencias a "PriceChanged" que realicen operaciones distintas a += o -= y las redirige al campo subyacente "priceChanged" del delegado.

Tercero, el compilador traduce las operaciones += y -= en el evento a llamadas a los accesores "add" y "remove" del evento. Curiosamente, esto hace que el comportamiento de += y -= sea único cuando se aplica a eventos: a diferencia de otros escenarios, no es simplemente un atajo para + y - seguido de una asignación.

Considera el siguiente ejemplo. La clase Stock dispara su evento PriceChanged cada vez que el precio de la acción cambia:

```
public delegate void PriceChangedHandler (decimal oldPrice,
                                          decimal newPrice);
public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) => this.symbol = symbol;

  public event PriceChangedHandler PriceChanged;

  public decimal Price
  {
    get => price;
    set
    {
      if (price == value) return;      // Exit if nothing has changed
      decimal oldPrice = price;
      price = value;
      if (PriceChanged != null)           // If invocation list not
        PriceChanged (oldPrice, price);   // empty, fire event.
    }
  }
}
```

Si eliminamos la palabra clave "event" de nuestro ejemplo, de manera que PriceChanged se convierte en un campo de delegado ordinario, obtendríamos los mismos resultados. Sin embargo, la clase Stock sería menos robusta, ya que los suscriptores podrían realizar las siguientes acciones para interferir entre sí:

- Reemplazar a otros suscriptores al reasignar PriceChanged (en lugar de utilizar el operador +=).
- Borrar todos los suscriptores (estableciendo PriceChanged a null).
- Transmitir a otros suscriptores al invocar el delegado.

Patrón de Evento Estándar

En casi todos los casos en los que se definen eventos en las bibliotecas de .NET, su definición sigue un patrón estándar diseñado para proporcionar consistencia en el código de la biblioteca y el código del usuario. En el núcleo del patrón de evento estándar se encuentra System.EventArgs, una clase predefinida de .NET sin miembros (excepto el campo estático Empty). EventArgs es una clase base para transmitir información sobre un evento. En nuestro ejemplo de Stock, podríamos crear una subclase de EventArgs para transmitir los precios antiguo y nuevo cuando se dispara el evento PriceChanged:

```
public class PriceChangedEventArgs : System.EventArgs
{
  public readonly decimal LastPrice;
  public readonly decimal NewPrice;

  public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice;
    NewPrice = newPrice;
  }
}
```

Para lograr la reutilización, la subclase de EventArgs se nombra según la información que contiene (en lugar del evento para el cual se utilizará). Típicamente expone los datos como propiedades o campos de solo lectura.

Una vez que se tiene la subclase de EventArgs, el siguiente paso es elegir o definir un delegado para el evento. Hay tres reglas:

- Debe tener un tipo de retorno void.
- Debe aceptar dos argumentos: el primero de tipo object y el segundo una subclase de EventArgs. El primer argumento indica el emisor del evento, y el segundo argumento contiene la información adicional a transmitir.
- Su nombre debe terminar con EventHandler.

.NET define un delegado genérico llamado System.EventHandler<> para ayudar con esto:

```
public delegate void EventHandler<TEventArgs> (object source, TEventArgs e)
```



Antes de que existieran los genéricos en el lenguaje (antes de C# 2.0), habríamos tenido que escribir un delegado personalizado de la siguiente manera:

```
public delegate void PriceChangedHandler
  (object sender, PriceChangedEventArgs e);
```

Por razones históricas, la mayoría de los eventos en las bibliotecas de .NET utilizan delegados definidos de esta manera.

El siguiente paso es definir un evento del tipo de delegado elegido. Aquí, utilizamos el delegado genérico EventHandler:

```
public class Stock
{
  ...
  public event EventHandler<PriceChangedEventArgs> PriceChanged;
}
```

Finalmente, el patrón requiere que escribas un método virtual protegido que dispare el evento. El nombre debe coincidir con el nombre del evento, precedido por la palabra "On", y luego aceptar un único argumento de tipo EventArgs:

```
public class Stock
{
  ...

  public event EventHandler<PriceChangedEventArgs> PriceChanged;

  protected virtual void OnPriceChanged (PriceChangedEventArgs e)
  {
    if (PriceChanged != null) PriceChanged (this, e);
  }
}
```



Para trabajar de manera robusta en escenarios multinucleo (Capítulo 14), es necesario asignar el delegado a una variable temporal antes de probarlo e invocarlo:

```
var temp = PriceChanged;
if (temp != null) temp (this, e);
```

Podemos lograr la misma funcionalidad sin la variable temp utilizando el operador de anulación condicional:

```
PriceChanged?.Invoke (this, e);
```

Siendo tanto seguro para subprocesos como conciso, esta es la mejor forma general de invocar eventos.

Esto proporciona un punto central desde el cual las subclases pueden invocar o anular el evento (asumiendo que la clase no está sellada).

Aquí tienes el ejemplo completo:

```
using System;

Stock stock = new Stock ("THPW");
stock.Price = 27.10M;
// Register with the PriceChanged event
stock.PriceChanged += stock_PriceChanged;
stock.Price = 31.59M;

void stock_PriceChanged (object sender, PriceChangedEventArgs e)
{
  if ((e.NewPrice - e.LastPrice) / e.LastPrice > 0.1M)
    Console.WriteLine ("Alert, 10% stock price increase!");
}

public class PriceChangedEventArgs : EventArgs
{
  public readonly decimal LastPrice;
  public readonly decimal NewPrice;

  public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice; NewPrice = newPrice;
  }
}

public class Stock

 {
   string symbol;
   decimal price;

   public Stock (string symbol) => this.symbol = symbol;

   public event EventHandler<PriceChangedEventArgs> PriceChanged;

   protected virtual void OnPriceChanged (PriceChangedEventArgs e)
   {
     PriceChanged?.Invoke (this, e);
   }

   public decimal Price
   {
     get => price;
     set
     {
       if (price == value) return;
       decimal oldPrice = price;
       price = value;
       OnPriceChanged (new PriceChangedEventArgs (oldPrice, price));
     }
   }
 }
```

El delegado predefinido no genérico EventHandler se puede utilizar cuando un evento no lleva información adicional. En este ejemplo, reescribimos la clase Stock de manera que el evento PriceChanged se dispare después de que cambie el precio y no se requiera ninguna información

adicional sobre el evento, aparte de que haya ocurrido. También hacemos uso de la propiedad EventArgs.Empty para evitar la creación innecesaria de una instancia de EventArgs:

```
public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) { this.symbol = symbol; }

  public event EventHandler PriceChanged;

  protected virtual void OnPriceChanged (EventArgs e)
  {
    PriceChanged?.Invoke (this, e);
  }

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      price = value;


        OnPriceChanged (EventArgs.Empty);
      }
    }
  }
```

Accesores de un evento

Los accesores de un evento son las implementaciones de sus funciones += y -=. Por defecto, los accesores se implementan implícitamente por el compilador. Considera esta declaración de evento:

```
public event EventHandler PriceChanged;
```

El compilador convierte esto en lo siguiente:

- Un campo de delegado privado.
- Un par público de funciones de acceso al evento (add_PriceChanged y remove_PriceChanged) cuyas implementaciones reenvían las operaciones += y -= al campo de delegado privado.

Puedes tomar el control de este proceso definiendo accesores de evento explícitos. Aquí tienes una implementación manual del evento PriceChanged de nuestro ejemplo anterior:

```
private EventHandler priceChanged;         // Declare a private delegate

public event EventHandler PriceChanged
{
  add    { priceChanged += value; }
  remove { priceChanged -= value; }
}
```

Este ejemplo es funcionalmente idéntico a la implementación predeterminada de accesores de C# (excepto que C# también garantiza la seguridad de subprocesos alrededor de la actualización del delegado mediante un algoritmo de comparación y intercambio sin bloqueo; consulta http://albahari.com/threading). Al definir accesores de evento explícitos, indicamos a C# que no genere la lógica predeterminada de campo y accesores.

Con accesores de evento explícitos, puedes aplicar estrategias más complejas para el almacenamiento y acceso del delegado subyacente. Hay tres escenarios en los que esto es útil:

- Cuando los accesores de evento son simplemente retransmisiones para otra clase que está transmitiendo el evento.
- Cuando la clase expone muchos eventos, para los cuales la mayoría de las veces hay muy pocos suscriptores, como un control de Windows. En tales casos, es mejor almacenar las instancias de delegado de los suscriptores en un diccionario, ya que un diccionario tendrá menos sobrecarga de almacenamiento que docenas de referencias de campo de delegado nulas.
- Cuando se implementa explícitamente una interfaz que declara un evento.

Aquí tienes un ejemplo que ilustra el último punto:

```
public interface IFoo { event EventHandler Ev; }

class Foo : IFoo
{
  private EventHandler ev;

  event EventHandler IFoo.Ev
  {
    add    { ev += value; }
    remove { ev -= value; }
  }
}
```


Las partes "add" y "remove" de un evento se compilan en métodos "add_XXX" y "remove_XXX".

Modificadores de Eventos

Al igual que los métodos, los eventos pueden ser virtuales, sobreescritos, abstractos o sellados. Los eventos también pueden ser estáticos:

```
public class Foo
{
  public static event EventHandler<EventArgs> StaticEvent;
  public virtual event EventHandler<EventArgs> VirtualEvent;
}
```

Expresiones Lambda

Una expresión lambda es un método sin nombre escrito en lugar de una instancia de delegado. El compilador convierte inmediatamente la expresión lambda en cualquiera de las siguientes opciones:

- Una instancia de delegado.
- Un árbol de expresiones, de tipo Expression<TDelegate>, que representa el código dentro de la expresión lambda en un modelo de objeto transitable. Esto permite interpretar la expresión lambda más tarde en tiempo de ejecución (ver "Construcción de expresiones de consulta" en la página 442).

En el siguiente ejemplo, x => x * x es una expresión lambda:

```
Transformer sqr = x => x * x;
Console.WriteLine (sqr(3));    // 9

delegate int Transformer (int i);
```



Internamente, el compilador resuelve las expresiones lambda de este tipo escribiendo un método privado y luego moviendo el código de la expresión a ese método.

Una expresión lambda tiene la siguiente forma:

```
(parameters) => expression-or-statement-block
```

Por conveniencia, puedes omitir los paréntesis solo si hay exactamente un parámetro de un tipo inferible.

En nuestro ejemplo, hay un solo parámetro, x, y la expresión es x * x:

```
x => x * x;
```

Cada parámetro de la expresión lambda corresponde a un parámetro del delegado, y el tipo de la expresión (que puede ser void) corresponde al tipo de retorno del delegado.

En nuestro ejemplo, x corresponde al parámetro i, y la expresión x * x corresponde al tipo de retorno int, por lo tanto, es compatible con el delegado Transformer:

```
delegate int Transformer (int i);
```

El código de una expresión lambda puede ser un bloque de instrucciones en lugar de una expresión. Podemos reescribir nuestro ejemplo de la siguiente manera:

```
x => { return x * x; };
```

Las expresiones lambda se utilizan con mayor frecuencia con los delegados Func y Action, por lo que es más común ver nuestra expresión anterior escrita de la siguiente manera:

```
Func<int,int> sqr = x => x * x;
```

Aquí tienes un ejemplo de una expresión que acepta dos parámetros:

```
Func<string,string,int> totalLength = (s1, s2) => s1.Length + s2.Length;
int total = totalLength ("hello", "world");   // total is 10;
```

Si no necesitas usar los parámetros, puedes descartarlos con un guion bajo (a partir de C# 9):

```
Func<string,string,int> totalLength = (_,_) => ...
```

Aquí tienes un ejemplo de una expresión que no toma argumentos:

```
Func<string> greetor = () => "Hello, world";
```

A partir de C# 10, el compilador permite la inferencia implícita con expresiones lambda que pueden resolverse a través de los delegados Func y Action, por lo que podemos acortar esta declaración a:

```
var greeter = () => "Hello, world";
```

Especificar explícitamente los tipos de parámetros y retorno de una expresión lambda

El compilador generalmente puede inferir el tipo de los parámetros de una expresión lambda contextualmente. Cuando esto no es posible, debes especificar explícitamente el tipo de cada parámetro. Considera los siguientes dos métodos:

```
void Foo<T> (T x)         {}
void Bar<T> (Action<T> a) {}
```

El siguiente código no se compilará porque el compilador no puede inferir el tipo de x:

```
Bar (x => Foo (x));     // What type is x?
```

Podemos solucionarlo especificando explícitamente el tipo de x de la siguiente manera:

```
Bar ((int x) => Foo (x));
```

Este ejemplo en particular es lo suficientemente simple como para que se pueda solucionar de otras dos formas:

```
Bar<int> (x => Foo (x));   // Specify type parameter for Bar
Bar<int> (Foo);            // As above, but with method group
```

El siguiente ejemplo ilustra otro uso para los tipos de parámetros explícitos (a partir de C# 10):

```
var sqr = (int x) => x * x;
```

El compilador infiere que sqr es del tipo Func<int, int>. (Sin especificar int, la inferencia implícita fallaría: el compilador sabría que sqr debería ser Func<T, T>, pero no sabría qué debe ser T).

A partir de C# 10, también puedes especificar el tipo de retorno de la lambda:

```
var sqr = int (int x) => x;
```

Especificar un tipo de retorno puede mejorar el rendimiento del compilador con lambdas anidadas complejas.

Capturar variables externas

Una expresión lambda puede hacer referencia a cualquier variable que sea accesible donde se define la expresión lambda. Estas se llaman variables externas y pueden incluir variables locales, parámetros y campos:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
Console.WriteLine (multiplier (3));           // 6
```

Las variables externas referenciadas por una expresión lambda se llaman variables capturadas. Una expresión lambda que captura variables se llama clausura (closure).



Las variables también pueden ser capturadas por métodos anónimos y métodos locales. Las reglas para las variables capturadas, en estos casos, son las mismas.

Las variables capturadas se evalúan cuando se invoca realmente el delegado, no cuando las variables fueron capturadas:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
factor = 10;
Console.WriteLine (multiplier (3));           // 30
```

Las expresiones lambda también pueden actualizar las variables capturadas:

```
int seed = 0;
Func<int> natural = () => seed++;
Console.WriteLine (natural());          // 0
Console.WriteLine (natural());          // 1
Console.WriteLine (seed);               // 2
```

Las variables capturadas tienen una vida útil extendida hasta la del delegado. En el siguiente ejemplo, la variable local `seed` normalmente desaparecería del ámbito cuando `Natural` termina de ejecutarse. Pero debido a que `seed` ha sido capturada, su vida útil se extiende hasta la del delegado que la captura, `natural`:

```
static Func<int> Natural()
{
  int seed = 0;
  return () => seed++;      // Returns a closure
}

static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural());      // 0
  Console.WriteLine (natural());      // 1
}
```

Una variable local instanciada dentro de una expresión lambda es única por cada invocación de la instancia del delegado. Si refactorizamos nuestro ejemplo anterior para instanciar `seed` dentro de la expresión lambda, obtenemos un resultado diferente (en este caso, indeseable):

```
static Func<int> Natural()
{
  return() => { int seed = 0; return seed++; };
}

static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural());          // 0
  Console.WriteLine (natural());          // 0
}
```



La captura se implementa internamente mediante la "elevación"(hoisting) de las variables capturadas a campos de una clase privada. Cuando se llama al método, la clase se instancia y se asocia a la instancia del delegado.

Lambdas estáticas

Cuando capturas variables locales, parámetros, campos de instancia o la referencia `this`, el compilador puede crear e instanciar una clase privada para almacenar una referencia a los datos capturados. Esto conlleva un pequeño costo de rendimiento, ya que se debe asignar memoria (y posteriormente recolectarla). En situaciones en las que el rendimiento es crítico, una estrategia de micro-optimización consiste en minimizar la carga en el recolector de basura asegurándose de que las rutas críticas del código incurran en pocas o ninguna asignación de memoria.

A partir de C# 9, puedes asegurarte de que una expresión lambda, una función local o un método anónimo no capturen estado aplicando la palabra clave static. Esto puede ser útil en escenarios de micro-optimización para evitar asignaciones de memoria no deseadas. Por ejemplo, podemos aplicar el modificador static a una expresión lambda de la siguiente manera:

```
Func<int, int> multiplier = static n => n * 2;
```

Si posteriormente intentamos modificar la expresión lambda para que capture una variable local, el compilador generará un error:

```
int factor = 2;
Func<int, int> multiplier = static n => n * factor;  // will not compile
```



La expresión lambda misma se evalúa como una instancia de delegado, lo cual requiere una asignación de memoria. Sin embargo, si la expresión lambda no captura variables,

el compilador reutilizará una única instancia almacenada en caché durante la vida de la aplicación, por lo que no habrá costo en la práctica.

Esta característica también se puede utilizar con funciones locales. En el siguiente ejemplo, el método Multiply no puede acceder a la variable factor:

```
void Foo()
{
  int factor = 123;
  static int Multiply (int x) => x * 2;   // Local static method
}
```

Por supuesto, el método Multiply aún podría asignar memoria explícitamente al llamar a new. Lo que esto nos protege es de una asignación de memoria inadvertida. Aplicar static aquí también es útil como herramienta de documentación, indicando un nivel reducido de acoplamiento.

Las expresiones lambda estáticas aún pueden acceder a variables estáticas y constantes (porque estas no requieren un cierre).



La palabra clave static actúa simplemente como una verificación; no tiene efecto en el IL que produce el compilador. Sin la palabra clave static, el compilador no genera un cierre a menos que sea necesario (y aún así, tiene trucos para mitigar el costo).

Capturando variables de iteración

Cuando capturas la variable de iteración de un bucle for, C# trata esa variable como si estuviera declarada fuera del bucle. Esto significa que la misma variable se captura en cada iteración. El siguiente programa escribe 333 en lugar de 012:

```
Action[] actions = new Action[3];

for (int i = 0; i < 3; i++)
  actions [i] = () => Console.Write (i);

foreach (Action a in actions) a();     // 333
```

Cada cierre (mostrado en negrita) captura la misma variable, `i`. Esto tiene sentido cuando consideras que `i` es una variable cuyo valor persiste entre las iteraciones del bucle; incluso puedes cambiar explícitamente `i` dentro del cuerpo del bucle si lo deseas. La consecuencia es que cuando los delegados se invocan más tarde, cada delegado ve el valor de `i` en el momento de la invocación, que es 3. Podemos ilustrarlo mejor expandiendo el bucle `for`, de la siguiente manera:

```
Action[] actions = new Action[3];
int i = 0;
actions[0] = () => Console.Write (i);
i = 1;
actions[1] = () => Console.Write (i);
i = 2;
actions[2] = () => Console.Write (i);
i = 3;
foreach (Action a in actions) a();    // 333
```

La solución, si queremos escribir 012, es asignar la variable de iteración a una variable local que tenga alcance dentro del bucle:

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
{
  int loopScopedi = i;
  actions [i] = () => Console.Write (loopScopedi);
}
foreach (Action a in actions) a();     // 012
```

Dado que `loopScopedi` se crea de nuevo en cada iteración, cada cierre captura una variable diferente.



Antes de C# 5.0, los bucles `foreach` funcionaban de la misma manera. Esto causaba confusión considerable: a diferencia de un bucle `for`, la variable de iteración en un bucle `foreach` es inmutable, por lo que esperarías que se tratara como local para el cuerpo del bucle. La buena noticia es que ahora está solucionado y puedes capturar de forma segura la variable de iteración de un bucle `foreach` sin sorpresas.

Expresiones lambda versus métodos locales

La funcionalidad de los métodos locales (consultar "Métodos locales" en la página 19) se superpone con la de las expresiones lambda. Los métodos locales tienen las siguientes tres ventajas:

- Pueden ser recursivos (pueden llamarse a sí mismos) sin necesidad de trucos engorrosos.
- Evitan el desorden de especificar un tipo de delegado.
- Incurren en una sobrecarga ligeramente menor.

Los métodos locales son más eficientes porque evitan la indirección de un delegado (lo cual cuesta algunos ciclos de CPU y una asignación de memoria). También pueden acceder a las variables locales del método contenedor sin que el compilador tenga que "elevar" las variables capturadas a una clase oculta.

Sin embargo, en muchos casos necesitas un delegado, especialmente cuando llamas a una función de orden superior, es decir, un método con un parámetro de tipo delegado. Por ejemplo:

```
public void Foo (Func<int,bool> predicate) { ... }
```

(Verás muchos más ejemplos de esto en el Capítulo 8). En estos casos, necesitas un delegado de todos modos, y es en estos casos precisamente donde las expresiones lambda suelen ser más concisas y limpias.

Métodos anónimos

Los métodos anónimos son una característica de C# 2.0 que fue en su mayoría reemplazada por las expresiones lambda de C# 3.0. Un método anónimo es similar a una expresión lambda, pero carece de las siguientes características:

- Parámetros de tipo implícito.
- Sintaxis de expresión (un método anónimo siempre debe ser un bloque de sentencias).
- La capacidad de compilar a un árbol de expresión, asignando a `Expression<T>`.

Un método anónimo utiliza la palabra clave `delegate` seguida (opcionalmente) de una declaración de parámetro y luego un cuerpo de método. Por ejemplo:

```
Transformer sqr = delegate (int x) {return x * x;};
Console.WriteLine (sqr(3));                          // 9

delegate int Transformer (int i);
```

La primera línea es semánticamente equivalente a la siguiente expresión lambda:

```
Transformer sqr =       (int x) => {return x * x;};
```

O simplemente:

```
Transformer sqr =            x  => x * x;
```

Los métodos anónimos capturan variables externas de la misma manera que las expresiones lambda y pueden ser precedidos por la palabra clave `static` para que se comporten como lambdas estáticas.



Una característica única de los métodos anónimos es que puedes omitir por completo la declaración de parámetros, incluso si el delegado lo espera. Esto puede ser útil al declarar eventos con un controlador predeterminado vacío:

```
public event EventHandler Clicked = delegate { };
```

Esto evita la necesidad de verificar si es nulo antes de disparar el evento. También es válido lo siguiente:

```
// Notice that we omit the parameters:
Clicked += delegate { Console.WriteLine ("clicked"); };
```

Declaraciones "try" y excepciones

Una declaración try especifica un bloque de código sujeto a manejo de errores o código de limpieza. El bloque try debe ir seguido de uno o más bloques catch y/o un bloque finally, o ambos. El bloque catch se ejecuta cuando se produce un error en el bloque try. El bloque finally se ejecuta después de que la ejecución abandone el bloque try (o, si está presente, el bloque catch) para realizar tareas de limpieza, independientemente de si se lanzó una excepción.

Un bloque catch tiene acceso a un objeto Exception que contiene información sobre el error. Utilizas un bloque catch para compensar el error o relanzar la excepción. Relanzas una excepción si simplemente quieres registrar el problema o si deseas lanzar una nueva excepción de un tipo de nivel superior.

Un bloque finally agrega determinismo a tu programa: el CLR se esfuerza por ejecutarlo siempre. Es útil para tareas de limpieza, como cerrar conexiones de red.

Una declaración try se ve así:

```
try
{
  ... // exception may get thrown within execution of this block
}
catch (ExceptionA ex)
{
  ... // handle exception of type ExceptionA
}
catch (ExceptionB ex)
{
  ... // handle exception of type ExceptionB
}
finally
{
  ... // cleanup code
}
```

Considera el siguiente programa:

```
int y = Calc (0);
Console.WriteLine (y);

int Calc (int x) => 10 / x;
```

Debido a que x es cero, el tiempo de ejecución lanza una excepción DivideByZeroException y nuestro programa termina. Podemos evitar esto capturando la excepción de la siguiente manera:

```
try
{
  int y = Calc (0);
  Console.WriteLine (y);
}
catch (DivideByZeroException ex)
{
  Console.WriteLine ("x cannot be zero");

}
Console.WriteLine ("program completed");

int Calc (int x) => 10 / x;
```

Aquí está la salida:

```
x cannot be zero
program completed
```



Este es un ejemplo simple para ilustrar el manejo de excepciones. Podríamos tratar este escenario particular de manera más efectiva en la práctica verificando explícitamente que el divisor no sea cero antes de llamar a `Calc`.

Es preferible verificar errores prevenibles en lugar de confiar en bloques try/catch, ya que las excepciones son relativamente costosas de manejar, tomando cientos de ciclos de reloj o más.

Cuando se lanza una excepción dentro de una declaración try, el CLR realiza una prueba:

¿La declaración try tiene algún bloque catch compatible?

- Si es así, la ejecución salta al bloque catch compatible, seguido del bloque finally (si está presente), y luego la ejecución continúa normalmente.
- Si no es así, la ejecución salta directamente al bloque finally (si está presente), luego el CLR busca bloques try en la pila de llamadas; si se encuentran, repite la prueba.

Si ninguna función en la pila de llamadas asume la responsabilidad de la excepción, el programa termina.

La cláusula catch

Una cláusula catch especifica qué tipo de excepción capturar. Esto debe ser `System.Exception` o una subclase de `System.Exception`.

Capturar `System.Exception` captura todos los posibles errores. Esto es útil en las siguientes circunstancias:

- Tu programa puede recuperarse potencialmente independientemente del tipo específico de excepción.
- Planeas relanzar la excepción (quizás después de registrarla).
- Tu manejador de errores es el último recurso antes de la terminación del programa.

Sin embargo, lo más común es capturar tipos de excepción específicos para evitar tener que lidiar con circunstancias para las cuales tu manejador no fue diseñado (por ejemplo, una `OutOfMemoryException`).

Puedes manejar múltiples tipos de excepción con múltiples cláusulas catch (una vez más, este ejemplo podría escribirse con verificación explícita de argumentos en lugar de manejo de excepciones):

```
class Test
{
  static void Main (string[] args)
  {
    try
    {
      byte b = byte.Parse (args[0]);
      Console.WriteLine (b);
    }
    catch (IndexOutOfRangeException)
    {
      Console.WriteLine ("Please provide at least one argument");
    }
    catch (FormatException)
    {
      Console.WriteLine ("That's not a number!");
    }
    catch (OverflowException)
    {
      Console.WriteLine ("You've given me more than a byte!");
    }
  }
}
```

Solo se ejecuta una cláusula catch para una excepción dada. Si deseas incluir una red de seguridad para capturar excepciones más generales (como System.Exception), debes colocar los controladores más específicos primero.

Se puede capturar una excepción sin especificar una variable si no necesitas acceder a sus propiedades:

```
catch (OverflowException)   // no variable
{
  ...
}
```

Además, puedes omitir tanto la variable como el tipo (lo que significa que se capturarán todas las excepciones):

```
catch { ... }
```

Filtros de excepción

Puedes especificar un filtro de excepción en una cláusula catch agregando una cláusula when:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
  ...
}
```

Si se produce una WebException en este ejemplo, se evalúa la expresión booleana que sigue a la palabra clave when. Si el resultado es falso, se ignora el bloque catch en cuestión y se consideran cualquier cláusula catch posterior. Con filtros de excepción, puede ser significativo capturar nuevamente el mismo tipo de excepción:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{ ... }
catch (WebException ex) when (ex.Status == WebExceptionStatus.SendFailure)
{ ... }
```

El bloque finally

Un bloque finally siempre se ejecuta, sin importar si se produce una excepción o si el bloque try se ejecuta hasta su finalización. Normalmente, se utilizan bloques finally para el código de limpieza.

Un bloque finally se ejecuta después de cualquiera de las siguientes situaciones:

- Se completa un bloque catch (o se lanza una nueva excepción).
- Se completa el bloque try (o se lanza una excepción para la cual no hay un bloque catch).
- El control abandona el bloque try debido a una instrucción de salto (por ejemplo, return o goto).

Las únicas cosas que pueden evitar que se ejecute un bloque finally son un bucle infinito o la finalización abrupta del proceso.

Un bloque finally ayuda a agregar determinismo a un programa. En el siguiente ejemplo, el archivo que abrimos siempre se cierra, independientemente de lo siguiente:

- El bloque try se completa de manera normal.
- La ejecución se devuelve antes debido a que el archivo está vacío (EndOfStream).
- Se lanza una IOException mientras se lee el archivo.

```
void ReadFile()
{
  StreamReader reader = null;    // In System.IO namespace
  try
  {
    reader = File.OpenText ("file.txt");
    if (reader.EndOfStream) return;
    Console.WriteLine (reader.ReadToEnd());
  }
  finally
  {
    if (reader != null) reader.Dispose();
  }
}
```

En este ejemplo, cerramos el archivo llamando a Dispose en el StreamReader. Llamar a Dispose en un objeto, dentro de un bloque finally, es una convención estándar y es compatible explícitamente en C# mediante la instrucción using.

La declaración using

Muchas clases encapsulan recursos no administrados, como identificadores de archivos, identificadores de gráficos o conexiones de bases de datos. Estas clases implementan System.IDisposable, que define un único método sin parámetros llamado Dispose para limpiar estos recursos. La declaración using proporciona una sintaxis elegante para llamar a Dispose en un objeto IDisposable dentro de un bloque finally.

Por lo tanto,

```
using (StreamReader reader = File.OpenText ("file.txt"))
{
  ...
}
```

es precisamente equivalente a lo siguiente:

```
{
  StreamReader reader = File.OpenText ("file.txt");
  try
  {
    ...
  }
  finally
  {
    if (reader != null)
      ((IDisposable)reader).Dispose();
  }
}
```

Declaraciones using

Si omites los corchetes y el bloque de instrucciones después de una declaración using (C# 8+), se convierte en una declaración using. El recurso se elimina cuando la ejecución sale del bloque de instrucciones que lo contiene:

```
if (File.Exists ("file.txt"))
{
  using var reader = File.OpenText ("file.txt");
  Console.WriteLine (reader.ReadLine());
  ...
}
```

En este caso, el lector se eliminará cuando la ejecución salga del bloque de instrucciones if.

Lanzamiento de excepciones

Las excepciones se pueden lanzar tanto por el tiempo de ejecución como en el código de usuario. En este ejemplo, Display lanza una System.ArgumentNullException:

```
try { Display (null); }
catch (ArgumentNullException ex)
{
  Console.WriteLine ("Caught the exception");


 }

 void Display (string name)
 {
   if (name == null)
     throw new ArgumentNullException (nameof (name));

   Console.WriteLine (name);
 }
```



Debido a que verificar nulos en un argumento y lanzar una ArgumentNullException es una ruta de código tan común, en realidad hay un atajo para ello a partir de .NET 6:

```
void Display (string name)
{
  ArgumentNullException.ThrowIfNull (name);
  Console.WriteLine (name);
}
```

Observa que no necesitamos especificar el nombre del parámetro. Explicaremos por qué más adelante, en "CallerArgumentExpression (C# 10)" en la página 231.

Expresiones throw

throw también puede aparecer como una expresión en funciones expresadas en cuerpo de expresión:

```
public string Foo() => throw new NotImplementedException();
```

Una expresión throw también puede aparecer en una expresión condicional ternaria:

```
string ProperCase (string value) =>
  value == null ? throw new ArgumentException ("value") :
  value == "" ? "" :
  char.ToUpper (value[0]) + value.Substring (1);
```

Volver a lanzar una excepción

Puedes capturar y volver a lanzar una excepción de la siguiente manera:

```
try {  ...  }
catch (Exception ex)
{
  // Log error
  ...
  throw;          // Rethrow same exception
}
```

 Si reemplazáramos throw con throw ex, el ejemplo seguiría funcionando, pero la propiedad StackTrace de la excepción propagada ya no reflejaría el error original.

Volver a lanzar de esta manera te permite registrar un error sin suprimirlo. También te permite retroceder en el manejo de una excepción si las circunstancias resultan ser más allá de lo esperado. El otro escenario común es volver a lanzar un tipo de excepción más específico:

```
try
{
  ... // Parse a DateTime from XML element data
}
catch (FormatException ex)
{
  throw new XmlException ("Invalid DateTime", ex);
}
```

Observa que al construir XmlException, pasamos la excepción original, ex, como segundo argumento. Este argumento popula la propiedad InnerException de la nueva excepción y ayuda en la depuración. Casi todos los tipos de excepción ofrecen un constructor similar.

Volver a lanzar una excepción menos específica es algo que podrías hacer al cruzar un límite de confianza, para no revelar información técnica a posibles hackers.

Propiedades clave de System.Exception

Las propiedades más importantes de System.Exception son las siguientes:

StackTrace

Una cadena que representa todos los métodos que se llaman desde el origen de la excepción hasta el bloque catch.

Message

Una cadena con una descripción del error.

InnerException

La excepción interna (si la hay) que causó la excepción externa. A su vez, esta excepción interna puede tener otra InnerException.



Todas las excepciones en C# son excepciones en tiempo de ejecución; no existe un equivalente a las excepciones comprobadas en tiempo de compilación de Java.

Tipos comunes de excepciones

Los siguientes tipos de excepciones se utilizan ampliamente en CLR (Common Language Runtime) y las bibliotecas .NET. Puedes lanzarlas tú mismo o usarlas como clases base para derivar tipos de excepciones personalizadas:

System.ArgumentException

Se lanza cuando se llama a una función con un argumento falso. Esto generalmente indica un error en el programa.

System.ArgumentNullException

Subclase de ArgumentException que se lanza cuando un argumento de función es (inesperadamente) nulo.

System.ArgumentOutOfRangeException

Subclase de ArgumentException que se lanza cuando un argumento (generalmente numérico) es demasiado grande o demasiado pequeño. Por ejemplo, se lanza cuando se pasa un número negativo a una función que solo acepta valores positivos.

System.InvalidOperationException

Se lanza cuando el estado de un objeto no es adecuado para que un método se ejecute correctamente, independientemente de los valores de los argumentos particulares. Ejemplos incluyen leer un archivo no abierto o obtener el siguiente elemento de un enumerador para el cual la lista subyacente ha sido modificada a mitad de la iteración.

System.NotSupportedException

Se lanza para indicar que una funcionalidad particular no es compatible. Un buen ejemplo es llamar al método Add en una colección para la cual IsReadOnly devuelve true.

System.NotImplementedException

Se lanza para indicar que una función aún no se ha implementado.

System.ObjectDisposedException

Se lanza cuando el objeto en el que se llama la función ha sido eliminado.

Otro tipo de excepción que se encuentra comúnmente es NullReferenceException. El CLR lanza esta excepción cuando intentas acceder a un miembro de un objeto cuyo valor es nulo (lo cual indica un error en tu código). Puedes lanzar una NullReferenceException directamente (con fines de prueba) de la siguiente manera:

```
throw null;
```

El patrón de método TryXXX

Al escribir un método, tienes la opción, cuando algo sale mal, de devolver algún tipo de código de error o lanzar una excepción. En general, se lanza una excepción cuando el error está fuera del flujo de trabajo normal, o si se espera que el llamador inmediato no pueda manejarlo. Sin embargo, en ocasiones, puede ser mejor ofrecer ambas opciones al consumidor. Un ejemplo de esto es el tipo int, que define dos versiones de su método Parse:

```
public int Parse     (string input);
public bool TryParse (string input, out int returnValue);
```

Si el análisis falla, Parse lanza una excepción; TryParse devuelve false.

Puedes implementar este patrón haciendo que el método XXX llame al método TryXXX de la siguiente manera:

```
public return-type XXX (input-type input)
{
  return-type returnValue;
  if (!TryXXX (input, out returnValue))
    throw new YYYException (...)

  return returnValue;
}
```

Alternativas a las excepciones

Al igual que con int.TryParse, una función puede comunicar un fallo enviando un código de error de regreso a la función que la llama a través de un tipo de retorno o un parámetro. Aunque esto puede funcionar con fallos simples y predecibles, se vuelve incómodo al extenderlo a todos los errores, contaminando las firmas de los métodos y creando una complejidad y desorden innecesarios. Además, no se puede generalizar a funciones que no son métodos, como operadores (por ejemplo, el operador de división) o propiedades. Una alternativa es colocar el error en un lugar común donde todas las funciones en la pila de llamadas puedan verlo (por ejemplo, un método estático que almacena el error actual por hilo). Sin embargo, esto requiere que cada función participe en un patrón de propagación de errores, que es engorroso y, irónicamente, propenso a errores.

Enumeración e iteradores

Enumeración

Un enumerador es un cursor de solo lectura y de avance único sobre una secuencia de valores. C# considera que un tipo es un enumerador si cumple alguna de las siguientes condiciones:

- Tiene un método público sin parámetros llamado MoveNext y una propiedad llamada Current.
- Implementa System.Collections.Generic.IEnumerator<T>.
- Implementa System.Collections.IEnumerator.

La instrucción foreach itera sobre un objeto enumerable. Un objeto enumerable es la representación lógica de una secuencia. No es en sí mismo un cursor, sino un objeto que produce cursores sobre sí mismo. C# considera que un tipo es enumerable si cumple alguna de las siguientes condiciones (la comprobación se realiza en este orden):

- Tiene un método público sin parámetros llamado GetEnumerator que devuelve un enumerador.
- Implementa System.Collections.Generic.IEnumerable<T>.
- Implementa System.Collections.IEnumerable.
- (A partir de C# 9) Puede enlazarse a un método de extensión llamado GetEnumerator que devuelve un enumerador (consulte "Métodos de extensión" en la página 202).

El patrón de enumeración es el siguiente:

```
class Enumerator   // Typically implements IEnumerator or IEnumerator<T>
{
  public IteratorVariableType Current { get {...} }
  public bool MoveNext() {...}


  }

  class Enumerable   // Typically implements IEnumerable or IEnumerable<T>
  {
    public Enumerator GetEnumerator() {...}
  }
```

Aquí está la forma de alto nivel de iterar a través de los caracteres en la palabra "beer" usando una instrucción foreach:

```
foreach (char c in "beer")
  Console.WriteLine (c);
```

Aquí está la forma de bajo nivel de iterar a través de los caracteres en "beer" sin usar una instrucción foreach:

```
using (var enumerator = "beer".GetEnumerator())
  while (enumerator.MoveNext())
  {
    var element = enumerator.Current;
    Console.WriteLine (element);
  }
```

Si el enumerador implementa IDisposable, la instrucción foreach también actúa como una instrucción using, liberando implícitamente el objeto enumerador.

El Capítulo 7 explica en más detalle las interfaces de enumeración.

Inicializadores de colecciones

Puedes instanciar y poblar un objeto enumerable en un solo paso:

```
using System.Collections.Generic;
...

List<int> list = new List<int> {1, 2, 3};
```

El compilador traduce esto a lo siguiente:

```
using System.Collections.Generic;
...

List<int> list = new List<int>();
list.Add (1);
list.Add (2);
list.Add (3);
```

Esto requiere que el objeto enumerable implemente la interfaz System.Collections.IEnumerable y que tenga un método Add que tenga el número apropiado de parámetros para la llamada. También puedes inicializar diccionarios de manera similar (ver "Diccionarios" en la página 372) de la siguiente manera:

```
var dict = new Dictionary<int, string>()
{
  { 5, "five" },



  { 10, "ten" }
};
```

De manera más concisa, puedes utilizar la siguiente sintaxis para inicializar un diccionario:

```
var dict = new Dictionary<int, string>()
{
  [3] = "three",
  [10] = "ten"
};
```

Esta sintaxis también es válida no solo para diccionarios, sino para cualquier tipo que tenga un indexador.

Iteradores

Mientras que una instrucción foreach consume un enumerador, un iterador es un productor de un enumerador. En este ejemplo, utilizamos un iterador para devolver una secuencia de números de Fibonacci (donde cada número es la suma de los dos anteriores):

```
using System;
using System.Collections.Generic;

foreach (int fib in Fibs(6))
  Console.Write (fib + "  ");
}

IEnumerable<int> Fibs (int fibCount)
{
  for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
  {
    yield return prevFib;
    int newFib = prevFib+curFib;
    prevFib = curFib;
    curFib = newFib;
  }
}

OUTPUT: 1  1  2  3  5  8
```

Mientras que una declaración de retorno expresa "Aquí está el valor que me pediste que devuelva de este método", una declaración de yield return expresa "Aquí está el siguiente elemento que me pediste que genere de este enumerador". En cada declaración de yield, se devuelve el control al llamador, pero el estado del método se mantiene para que pueda continuar ejecutándose tan pronto como el llamador enumere el siguiente elemento. El tiempo de vida de este estado está vinculado al enumerador, de modo que el estado se puede liberar cuando el llamador haya terminado de enumerar.



El compilador convierte los métodos iteradores en clases privadas que implementan IEnumerable<T> y/o IEnumerator<T>. La lógica dentro del bloque del iterador se "invierte" y se inserta en el método MoveNext y en la propiedad Current de la clase enumeradora generada por el compilador. Esto significa que cuando llamas a un método iterador, lo único que haces es instanciar la clase generada por el compilador; ¡ninguno de tu código se ejecuta realmente! Tu código se ejecuta solo cuando comienzas a enumerar la secuencia resultante, típicamente con una instrucción foreach.

Los iteradores pueden ser métodos locales (ver "Métodos locales" en la página 19).

Semántica de los iteradores

Un iterador es un método, propiedad o indexador que contiene una o más declaraciones yield. Un iterador debe devolver una de las siguientes cuatro interfaces (de lo contrario, el compilador generará un error):

```
// Enumerable interfaces
System.Collections.IEnumerable
System.Collections.Generic.IEnumerable<T>

// Enumerator interfaces
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

Un iterador tiene diferentes semánticas dependiendo de si devuelve una interfaz enumerable o una interfaz enumeradora. Esto se describe en el Capítulo 7.

Se permiten múltiples declaraciones yield:

```
foreach (string s in Foo())
  Console.WriteLine(s);         // Prints "One","Two","Three"

IEnumerable<string> Foo()
{
  yield return "One";
  yield return "Two";
  yield return "Three";
}
```

yield break

Una declaración de return es ilegal en un bloque de iterador; en su lugar, debes utilizar la declaración yield break para indicar que el bloque del iterador debe salir temprano, sin devolver más elementos. Podemos modificar Foo de la siguiente manera para demostrarlo:

```
IEnumerable<string> Foo (bool breakEarly)
{
  yield return "One";
  yield return "Two";

  if (breakEarly)
    yield break;

  yield return "Three";
}
```

Los iteradores y los bloques try/catch/finally

Una instrucción yield return no puede aparecer en un bloque try que tenga una cláusula catch:

```
IEnumerable<string> Foo()
{
  try { yield return "One"; }    // Illegal
  catch { ... }
}
```

Tampoco puede aparecer yield return en un bloque catch o finally. Estas restricciones se deben a que el compilador debe traducir los iteradores en clases ordinarias con los miembros MoveNext, Current y Dispose, y traducir bloques de manejo de excepciones crearía una complejidad excesiva.

Sin embargo, puedes utilizar yield dentro de un bloque try que tenga (solamente) un bloque finally:

```
IEnumerable<string> Foo()
{
  try { yield return "One"; }    // OK
  finally { ... }
}
```

El código en el bloque finally se ejecuta cuando el enumerador consumidor alcanza el final de la secuencia o se desecha. Una instrucción foreach implícitamente desecha el enumerador si se rompe prematuramente, lo que hace que esta sea una forma segura de consumir enumeradores.

Cuando trabajas explícitamente con enumeradores, un problema es abandonar la enumeración prematuramente sin deshacerse de ella, evitando así el bloque finally. Puedes evitar este riesgo envolviendo el uso explícito de enumeradores en una instrucción using:

```
string firstElement = null;
var sequence = Foo();
using (var enumerator = sequence.GetEnumerator())
  if (enumerator.MoveNext())
    firstElement = enumerator.Current;
```

Composición de secuencias

Los iteradores son altamente componibles. Podemos extender nuestro ejemplo, esta vez para mostrar solo números de Fibonacci pares:

```
using System;
using System.Collections.Generic;

foreach (int fib in EvenNumbersOnly (Fibs(6)))
  Console.WriteLine (fib);

IEnumerable<int> Fibs (int fibCount)
{
  for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)

  {
    yield return prevFib;
    int newFib = prevFib+curFib;
    prevFib = curFib;
    curFib = newFib;
  }
}

IEnumerable<int> EvenNumbersOnly (IEnumerable<int> sequence)
{
  foreach (int x in sequence)
    if ((x % 2) == 0)
      yield return x;
}
```

Cada elemento no se calcula hasta el último momento, es decir, cuando se solicita mediante una operación MoveNext(). La figura 4-1 muestra las solicitudes de datos y la salida de datos a lo largo del tiempo.

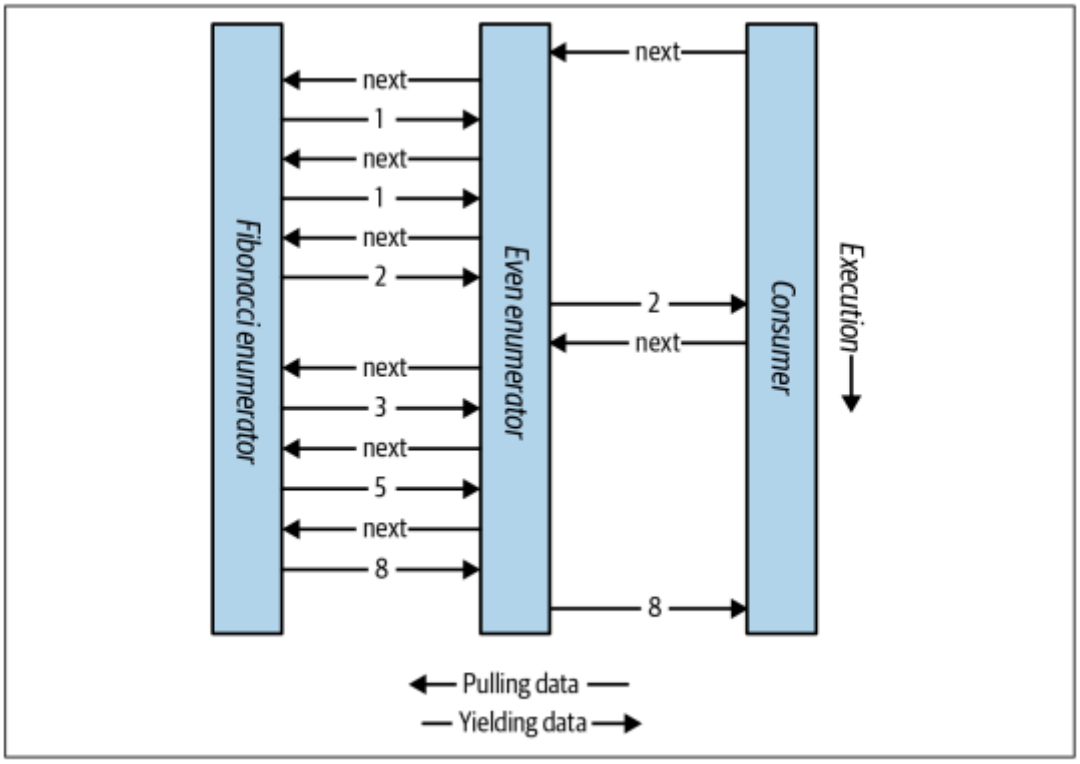


Figure 4-1. Composing sequences

La capacidad de componer el patrón del iterador es extremadamente útil en LINQ; discutiremos el tema nuevamente en el Capítulo 8.

Tipos de valor anulables

Los tipos de referencia pueden representar un valor inexistente con una referencia nula. Sin embargo, los tipos de valor no pueden representar de forma ordinaria valores nulos:

```
string s = null;       // OK, Reference Type
int i = null;          // Compile Error, Value Type cannot be null
```

Para representar un valor nulo en un tipo de valor, debes utilizar una construcción especial llamada tipo nullable. Un tipo nullable se denota con un tipo de valor seguido del símbolo "?".

```
int? i = null;                  // OK, Nullable Type
Console.WriteLine (i == null);  // True
```

La estructura Nullable<T>

El tipo T? se traduce en System.Nullable<T>, que es una estructura ligera e inmutable que tiene solo dos campos para representar el valor y si tiene un valor o no. La esencia de System.Nullable<T> es muy simple:

```
public struct Nullable<T> where T : struct
{
  public T Value {get;}
  public bool HasValue {get;}
  public T GetValueOrDefault();
  public T GetValueOrDefault (T defaultValue);
  ...
}
```

El código

```
int? i = null;
Console.WriteLine (i == null);              // True
```

se traduce a lo siguiente:

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine (! i.HasValue);           // True
```

Intentar obtener el valor de Value cuando HasValue es false genera una excepción InvalidOperationException. GetValueOrDefault() devuelve el valor de Value si HasValue es true; de lo contrario, devuelve new T() o un valor predeterminado personalizado especificado.

El valor predeterminado de T? es null.

Conversiones implícitas y explícitas de valores nulos

La conversión de T a T? es implícita, mientras que la conversión de T? a T es explícita:

```
int? x = 5;        // implicit
int y = (int)x;    // explicit
```

La conversión explícita es equivalente a llamar directamente a la propiedad Value del objeto nullable. Por lo tanto, se genera una excepción InvalidOperationException si HasValue es false.

Boxing y unboxing de valores nulos

Cuando se realiza el boxing de T?, el valor en el heap contiene T, no T?. Esta optimización es posible porque un valor en el heap es un tipo de referencia que ya puede expresar null.

C# también permite el unboxing de tipos de valor nulos con el operador 'as'. El resultado será nulo si la conversión falla:

```
object o = "string";
int? x = o as int?;
Console.WriteLine (x.HasValue);   // False
```

Elevación de Operadores(Operator Lifting)

La estructura Nullable<T> no define operadores como <, > o incluso ==. A pesar de esto, el siguiente código se compila y se ejecuta correctamente:

```
int? x = 5;
int? y = 10;
bool b = x < y;      // true
```

Esto funciona porque el compilador toma prestado o "eleva" el operador de menor que de tipo de valor subyacente. Semánticamente, traduce la expresión de comparación anterior en esto:

```
bool b = (x.HasValue && y.HasValue) ? (x.Value < y.Value) : false;
```

En otras palabras, si tanto x como y tienen valores, los compara mediante el operador menor que de int; de lo contrario, devuelve false.

La elevación de operadores significa que puedes usar implícitamente los operadores de T en T?. Puedes definir operadores para T? con el fin de proporcionar un comportamiento especializado para nulos, pero en la gran mayoría de los casos, es mejor confiar en que el compilador aplique automáticamente la lógica sistemática de nulos por ti. Aquí tienes algunos ejemplos:

```
int? x = 5;
int? y = null;

// Equality operator examples
Console.WriteLine (x == y);    // False
Console.WriteLine (x == null); // False
Console.WriteLine (x == 5);    // True
Console.WriteLine (y == null); // True
Console.WriteLine (y == 5);    // False
Console.WriteLine (y != 5);    // True

// Relational operator examples
Console.WriteLine (x < 6);     // True
Console.WriteLine (y < 6);     // False
Console.WriteLine (y > 6);     // False

// All other operator examples
Console.WriteLine (x + 5);     // 10
Console.WriteLine (x + y);     // null (prints empty line)
```

El compilador realiza la lógica de nulos de manera diferente según la categoría del operador. Las siguientes secciones explican estas reglas diferentes.

Operadores de igualdad (== y !=)

Los operadores de igualdad elevados manejan los nulos de la misma manera que lo hacen los tipos de referencia. Esto significa que dos valores nulos son iguales:

```
Console.WriteLine (       null ==        null);   // True
Console.WriteLine ((bool?)null == (bool?)null);   // True
```

Además:

- Si exactamente uno de los operandos es nulo, los operandos son desiguales.
- Si ambos operandos no son nulos, se comparan sus valores.

Operadores relacionales (<, <=, >=, >)

Los operadores relacionales funcionan bajo el principio de que no tiene sentido comparar operandos nulos. Esto significa que comparar un valor nulo con otro valor nulo o no nulo devuelve false:

```
bool b = x < y;    // Translation:

bool b = (x.HasValue && y.HasValue)
         ? (x.Value < y.Value)
         : false;

// b is false (assuming x is 5 and y is null)
```

Todos los demás operadores (+, -, *, /, %, &, |, ^, <<, >>, +, ++, --, !, ~)

Estos operadores devuelven nulo cuando alguno de los operandos es nulo. Este patrón debería resultar familiar para los usuarios de SQL:

```
int? c = x + y;   // Translation:

int? c = (x.HasValue && y.HasValue)
         ? (int?) (x.Value + y.Value)
         : null;

// c is null (assuming x is 5 and y is null)
```

Una excepción ocurre cuando los operadores & y | se aplican a bool?, de lo cual hablaremos en breve.

Mezcla de operadores nulos y no nulos

Puedes combinar y mezclar tipos de valor nulos y no nulos (esto funciona porque hay una conversión implícita de T a T?):

```
int? a = null;
int b = 2;
int? c = a + b;   // c is null - equivalent to a + (int?)b
```

bool? con los operadores & y |

Cuando se suministran operandos de tipo bool?, los operadores & y | tratan a null como un valor desconocido. Por lo tanto, null | true es true porque:

- Si el valor desconocido es false, el resultado sería true.
- Si el valor desconocido es true, el resultado sería true.

De manera similar, null & false es false. Este comportamiento debería ser familiar para los usuarios de SQL. El siguiente ejemplo muestra otras combinaciones:

```
bool? n = null;
bool? f = false;
bool? t = true;
Console.WriteLine (n | n);    // (null)
Console.WriteLine (n | f);    // (null)
Console.WriteLine (n | t);    // True
Console.WriteLine (n & n);    // (null)
Console.WriteLine (n & f);    // False
Console.WriteLine (n & t);    // (null)
```

Tipos de valor nulos y operadores nulos

Los tipos de valor nulos funcionan especialmente bien con el operador ?? (consultar el operador de fusión nula en la página 74), como se ilustra en este ejemplo:

```
int? x = null;
int y = x ?? 5;        // y is 5

int? a = null, b = 1, c = 2;
Console.WriteLine (a ?? b ?? c);  // 1 (first non-null value)
```

Usar ?? en un tipo de valor nulo es equivalente a llamar a GetValueOrDefault con un valor predeterminado explícito, excepto que la expresión para el valor predeterminado no se evalúa si la variable no es nula.

Los tipos de valor nulos también funcionan bien con el operador de condición nula (consultar el operador condicional nulo en la página 75). En el siguiente ejemplo, length se evalúa como null:

```
System.Text.StringBuilder sb = null;
int? length = sb?.ToString().Length;
```

Podemos combinar esto con el operador de fusión nula para evaluarlo como cero en lugar de nulo:

```
int length = sb?.ToString().Length ?? 0;  // Evaluates to 0 if sb is null
```

Escenarios para tipos de valor nulos

Uno de los escenarios más comunes para los tipos de valor nulos es representar valores desconocidos. Esto ocurre con frecuencia en la programación de bases de datos, donde una clase se asigna a una tabla con columnas nulas. Si estas columnas son de tipo string (por ejemplo, una columna EmailAddress en una tabla Customer), no hay problema porque string es un tipo de referencia en el CLR, que puede ser nulo. Sin embargo, la mayoría de los otros tipos de columna de SQL se asignan a tipos de estructura CLR, lo que hace que los tipos de valor nulos sean muy útiles al mapear SQL al CLR:

```
// Maps to a Customer table in a database
public class Customer
{
  ...
  public decimal? AccountBalance;
}
```

Un tipo nulo también se puede utilizar para representar el campo de respaldo de lo que a veces se llama una propiedad ambiental. Una propiedad ambiental, si es nula, devuelve el valor de su elemento principal:

```
public class Row
{
  ...
  Grid parent;
  Color? color;

  public Color Color
  {
    get { return color ?? parent.Color; }
    set { color = value == parent.Color ? (Color?)null : value; }
  }
}
```

Alternativas a los tipos de valor nulos

Antes de que los tipos de valor nulos fueran parte del lenguaje C# (es decir, antes de C# 2.0), existían muchas estrategias para lidiar con ellos, ejemplos de los cuales todavía aparecen en las bibliotecas .NET por razones históricas. Una de estas estrategias es designar un valor no nulo en particular como el "valor nulo"; un ejemplo está en las clases string y array. String.IndexOf devuelve el valor mágico de -1 cuando no se encuentra el carácter:

```
int i = "Pink".IndexOf ('b');
Console.WriteLine (i);         // -1
```

Sin embargo, Array.IndexOf devuelve -1 solo si el índice es acotado a 0. La fórmula más general es que IndexOf devuelve uno menos que el límite inferior del array. En el siguiente ejemplo, IndexOf devuelve 0 cuando no se encuentra un elemento:

```
// Create an array whose lower bound is 1 instead of 0:

Array a = Array.CreateInstance (typeof (string),
                                new int[] {2}, new int[] {1});
a.SetValue ("a", 1);
a.SetValue ("b", 2);
Console.WriteLine (Array.IndexOf (a, "c"));  // 0
```

Designar un "valor mágico" presenta varios problemas:

- Significa que cada tipo de valor tiene una representación diferente de null. En contraste, los tipos de valor nulos proporcionan un patrón común que funciona para todos los tipos de valor.
- Puede que no haya un valor designado razonable. En el ejemplo anterior, -1 no siempre se podría utilizar. Lo mismo ocurre con nuestro ejemplo anterior que representa un saldo de cuenta desconocido.
- Olvidarse de probar el valor "mágico" resulta en un valor incorrecto que podría pasar desapercibido hasta más adelante en la ejecución, cuando realiza un truco de magia no deseado. Sin embargo, olvidarse de probar HasValue en un valor nulo arroja una InvalidOperationException de inmediato.

- La capacidad de que un valor sea nulo no se captura en el tipo. Los tipos comunican la intención de un programa, permiten que el compilador verifique la corrección y habilitan un conjunto coherente de reglas impuestas por el compilador.

Tipos de referencia nulos

Mientras que los tipos de valor nulos llevan la nulabilidad a los tipos de valor, los tipos de referencia nulos (C# 8+) hacen lo contrario. Cuando están habilitados, brindan (en cierto grado) no nulabilidad a los tipos de referencia, con el propósito de ayudar a evitar NullReferenceExceptions.

Los tipos de referencia nulos introducen un nivel de seguridad que se aplica puramente mediante el compilador, en forma de advertencias cuando detecta código que corre el riesgo de generar una NullReferenceException.

Para habilitar los tipos de referencia nulos, debes agregar el elemento Nullable a tu archivo de proyecto .csproj (si deseas habilitarlo para todo el proyecto):

```
<PropertyGroup>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

y/o utilizar las siguientes directivas en tu código, en los lugares donde deben tener efecto:

```
#nullable enable   // enables nullable reference types from this point on
#nullable disable  // disables nullable reference types from this point on
#nullable restore  // resets nullable reference types to project setting
```

Después de habilitarlos, el compilador establece la no nulabilidad como predeterminada: si deseas que un tipo de referencia acepte nulos sin que el compilador genere una advertencia, debes aplicar el sufijo ? para indicar un tipo de referencia nulo. En el siguiente ejemplo, s1 no es nulo, mientras que s2 es nulo:

```
#nullable enable    // Enable nullable reference types

string s1 = null;   // Generates a compiler warning!
string? s2 = null;  // OK: s2 is nullable reference type
```



Los tipos de referencia nulos no introducen diferencias en tiempo de ejecución entre `string` y `string?` porque son constructos de tiempo de compilación. En contraste, los tipos de valor nulos introducen algo concreto en el sistema de tipos, es decir, la estructura `Nullable<T>`.

Si tienes una clase como la siguiente, que contiene un campo de tipo `string`, generará una advertencia si el campo no se inicializa:

```
class Foo { string x; }
```

La advertencia desaparecerá si inicializas el campo, ya sea mediante un inicializador de campo o mediante código en el constructor.

El operador de nullificación

El compilador también te advertirá al desreferenciar un tipo de referencia nulo si considera que podría producirse una `NullReferenceException`. En el siguiente ejemplo, el acceso a la propiedad `Length` de la cadena genera una advertencia:

```
void Foo (string? s) => Console.Write (s.Length);
```

Puedes eliminar la advertencia con el operador de nullificación `!`:

```
void Foo (string? s) => Console.Write (s!.Length);
```

Sin embargo, el uso del operador de nullificación en este ejemplo es peligroso, ya que podríamos terminar lanzando la misma `NullReferenceException` que estábamos tratando de evitar en primer lugar. Podemos corregirlo de la siguiente manera:

```
void Foo (string? s)
{
  if (s != null) Console.Write (s.Length);
}
```

Observa que ahora no necesitamos el operador de nullificación. Esto se debe a que el compilador realiza un análisis de flujo estático y es lo suficientemente inteligente como para inferir, al menos en casos simples, cuándo una desreferencia es segura y no hay posibilidad de una `NullReferenceException`.

La capacidad del compilador para detectar y advertir no es infalible y también existen límites en cuanto a lo que es posible en términos de cobertura. Por ejemplo, el compilador no puede saber si los elementos de un arreglo han sido poblados, por lo que el siguiente código no genera una advertencia:

```
var strings = new string[10];
Console.WriteLine (strings[0].Length);
```

Separación de las anotaciones y contextos de advertencia

Al habilitar los tipos de referencia nulos a través de la directiva `#nullable enable` (o la configuración de proyecto `<Nullable>enable</Nullable>`), se realizan dos acciones:

- Habilita el contexto de anotaciones nulas, lo cual indica al compilador que trate todas las declaraciones de variables de tipo de referencia como no anulables a menos que se les agregue el símbolo "?" al final.

- Habilita el contexto de advertencias de nulabilidad, lo cual indica al compilador que genere advertencias al encontrar código en riesgo de generar una excepción de referencia nula.

A veces puede ser útil separar estos dos conceptos y habilitar solo el contexto de anotaciones o, de manera menos útil, solo el contexto de advertencias:

```
#nullable enable annotations    // Enable the annotation context
// OR:
#nullable enable warnings       // Enable the warning context
```

(El mismo truco funciona con #nullable disable y #nullable restore).

También puedes hacerlo a través del archivo del proyecto:

```
<Nullable>annotations</Nullable>
<!-- OR -->
<Nullable>warnings</Nullable>
```

Habilitar solo el contexto de anotaciones para una clase o ensamblado en particular puede ser un buen primer paso para introducir tipos de referencia nulos en una base de código existente. Al anotar correctamente los miembros públicos, tu clase o ensamblado puede actuar como un "buen ciudadano" para otras clases o ensamblados, permitiéndoles beneficiarse completamente de los tipos de referencia nulos sin tener que lidiar con advertencias en tu propia clase o ensamblado.

Tratamiento de advertencias de nulabilidad como errores

En proyectos nuevos, tiene sentido habilitar completamente el contexto de nulabilidad desde el principio. Es posible que desees dar el paso adicional de tratar las advertencias de nulabilidad como errores para que tu proyecto no pueda compilar hasta que se resuelvan todas las advertencias de nulidad:

```
<PropertyGroup>
  <Nullable>enable</Nullable>
  <WarningsAsErrors>CS8600;CS8602;CS8603</WarningsAsErrors>
</PropertyGroup>
```

Métodos de extensión

Los métodos de extensión permiten extender un tipo existente con nuevos métodos sin alterar la definición del tipo original. Un método de extensión es un método estático de una clase estática, donde se aplica el modificador "this" al primer parámetro. El tipo del primer parámetro será el tipo que se está extendiendo:

```
public static class StringHelper
{
  public static bool IsCapitalized (this string s)
  {

    if (string.IsNullOrEmpty(s)) return false;
    return char.IsUpper (s[0]);
  }
}
```

El método de extensión IsCapitalized se puede llamar como si fuera un método de instancia en un objeto de tipo string, de la siguiente manera:

```
Console.WriteLine ("Perth".IsCapitalized());
```

Una llamada a un método de extensión, cuando se compila, se traduce de vuelta a una llamada ordinaria a un método estático:

```
Console.WriteLine (StringHelper.IsCapitalized ("Perth"));
```

La traducción funciona de la siguiente manera:

```
arg0.Method (arg1, arg2, ...);              // Extension method call
StaticClass.Method (arg0, arg1, arg2, ...); // Static method call
```

Las interfaces también pueden ser extendidas:

```
public static T First<T> (this IEnumerable<T> sequence)
{
  foreach (T element in sequence)
    return element;

  throw new InvalidOperationException ("No elements!");
}
...
Console.WriteLine ("Seattle".First());   // S
```

Encadenamiento de Métodos de Extensión

Los métodos de extensión, al igual que los métodos de instancia, proporcionan una forma ordenada de encadenar funciones. Considera las siguientes dos funciones:

```
public static class StringHelper
{
  public static string Pluralize (this string s) {...}
  public static string Capitalize (this string s) {...}
}
```

Tanto x como y son equivalentes y ambos evalúan a "Sausages", pero x utiliza métodos de extensión, mientras que y utiliza métodos estáticos:

```
string x = "sausage".Pluralize().Capitalize();
string y = StringHelper.Capitalize (StringHelper.Pluralize ("sausage"));
```

Ambigüedad y Resolución

Namespaces

Los métodos de extensión solo pueden ser accedidos si la clase que los define está en el ámbito, típicamente mediante la importación de su espacio de nombres. Considera el método de extensión IsCapitalized en el siguiente ejemplo:

```
using System;

namespace Utils
{
  public static class StringHelper
  {
    public static bool IsCapitalized (this string s)
    {
      if (string.IsNullOrEmpty(s)) return false;
      return char.IsUpper (s[0]);
    }
  }
}
```

Para usar IsCapitalized, la siguiente aplicación debe importar Utils para evitar un error en tiempo de compilación:

```
namespace MyApp
{
  using Utils;

  class Test
  {
    static void Main() => Console.WriteLine ("Perth".IsCapitalized());
  }
}
```

Métodos de extensión versus métodos de instancia

Cualquier método de instancia compatible siempre tendrá prioridad sobre un método de extensión. En el siguiente ejemplo, el método Foo de la clase Test siempre tendrá prioridad, incluso cuando se llame con un argumento x de tipo int:

```
class Test
{
  public void Foo (object x) { }    // This method always wins
}

static class Extensions
{
  public static void Foo (this Test t, int x) { }
}
```

La única forma de llamar al método de extensión en este caso es mediante la sintaxis estática normal, es decir, Extensions.Foo(...).

Métodos de extensión versus métodos de extensión

Si dos métodos de extensión tienen la misma firma, el método de extensión debe ser llamado como un método estático normal para desambiguar el método a llamar. Sin embargo, si un método de extensión tiene argumentos más específicos, el método más específico tiene prioridad.

Para ilustrar, considera las siguientes dos clases:

```
static class StringHelper
{
  public static bool IsCapitalized (this string s) {...}
}
static class ObjectHelper
{
  public static bool IsCapitalized (this object s) {...}
}
```

El siguiente código llama al método IsCapitalized de StringHelper:

```
bool test1 = "Perth".IsCapitalized();
```

Las clases y las estructuras se consideran más específicas que las interfaces.

Rebajar un método de extensión(Demoting an extensión method)

Puede surgir un escenario interesante cuando Microsoft agrega un método de extensión a una biblioteca de tiempo de ejecución de .NET que entra en conflicto con un método de extensión en alguna biblioteca de terceros existente. Como autor de la biblioteca de terceros, es posible que desees "retirar" tu método de extensión, pero sin eliminarlo y sin romper la compatibilidad binaria con los consumidores existentes.

Afortunadamente, esto se puede lograr fácilmente simplemente eliminando la palabra clave `this` de la definición de tu método de extensión. Esto rebaja tu método de extensión a un método estático ordinario. La belleza de esta solución es que cualquier ensamblado que se haya compilado con tu antigua biblioteca seguirá funcionando (y se vinculará a tu método, como antes). Esto se debe a que las llamadas a métodos de extensión se convierten en llamadas a métodos estáticos durante la compilación.

Los consumidores se verán afectados por tu rebaja solo cuando vuelvan a compilar, en ese momento las llamadas a tu antiguo método de extensión se vincularán ahora a la versión de Microsoft (si se ha importado el espacio de nombres). Si el consumidor todavía desea llamar a tu método, pueden hacerlo invocándolo como un método estático.

Tipos anónimos

Un tipo anónimo es una clase simple creada por el compilador sobre la marcha para almacenar un conjunto de valores. Para crear un tipo anónimo, utiliza la palabra clave `new` seguida de un inicializador de objetos, especificando las propiedades y valores que contendrá el tipo; por ejemplo:

```
var dude = new { Name = "Bob", Age = 23 };
```

El compilador traduce esto a (aproximadamente) lo siguiente:

```
internal class AnonymousGeneratedTypeName
{
  private string name;  // Actual field name is irrelevant
  private int    age;   // Actual field name is irrelevant

  public AnonymousGeneratedTypeName (string name, int age)
  {
    this.name = name; this.age = age;


  }

  public string  Name { get { return name; } }
  public int     Age  { get { return age;  } }

  // The Equals and GetHashCode methods are overridden (see Chapter 6).
  // The ToString method is also overridden.
}
...

var dude = new AnonymousGeneratedTypeName ("Bob", 23);
```

Debes usar la palabra clave `var` para hacer referencia a un tipo anónimo porque no tiene un nombre.

El nombre de la propiedad de un tipo anónimo se puede inferir a partir de una expresión que es en sí un identificador (o termina con uno); por lo tanto,

```
int Age = 23;
var dude = new { Name = "Bob", Age, Age.ToString().Length };
```

es equivalente a lo siguiente:

```
var dude = new { Name = "Bob", Age = Age, Length = Age.ToString().Length };
```

Dos instancias de tipos anónimos declaradas en la misma asamblea tendrán el mismo tipo subyacente si sus elementos tienen los mismos nombres y tipos:

```
var a1 = new { X = 2, Y = 4 };
var a2 = new { X = 2, Y = 4 };
Console.WriteLine (a1.GetType() == a2.GetType());   // True
```

Además, el método Equals está sobrescrito para realizar una comparación de igualdad estructural (comparación de los datos):

```
Console.WriteLine (a1.Equals (a2));   // True
```

mientras que el operador de igualdad (==) realiza una comparación referencial:

```
Console.WriteLine (a1 == a2);         // False
```

Puedes crear matrices de tipos anónimos de la siguiente manera:

```
var dudes = new[]
{
  new { Name = "Bob", Age = 30 },
  new { Name = "Tom", Age = 40 }
};
```

Un método no puede devolver un objeto de tipo anónimo (útilmente) porque es ilegal escribir un método cuyo tipo de retorno es `var`:

```
var Foo() => new { Name = "Bob", Age = 30 };  // Not legal!
```

En su lugar, debes usar `object` o `dynamic`, y luego quien llame a `Foo` debe confiar en la asignación dinámica, con pérdida de seguridad de tipos estáticos (y IntelliSense en Visual Studio):

```
dynamic Foo() => new { Name = "Bob", Age = 30 };  // No static type safety.
```

Los tipos anónimos son inmutables, por lo que las instancias no se pueden modificar después de su creación. Sin embargo, a partir de C# 10, puedes usar la palabra clave `with` para crear una copia con variaciones (mutación no destructiva):

```
var a1 = new { A = 1, B = 2, C = 3, D = 4, E = 5 };
var a2 = a1 with { E = 10 };
Console.WriteLine (a2);      // { A = 1, B = 2, C = 3, D = 4, E = 10 }
```

Los tipos anónimos son particularmente útiles al escribir consultas LINQ (ver Capítulo 8).

Tuplas (Tuples)

Las tuplas, al igual que los tipos anónimos, proporcionan una forma sencilla de almacenar un conjunto de valores. El objetivo principal de las tuplas es devolver de forma segura múltiples valores desde un método sin recurrir a parámetros de salida (algo que no se puede hacer con tipos anónimos).



Las tuplas hacen casi todo lo que los tipos anónimos hacen y más. Su única desventaja, como verás pronto, es la eliminación del tipo en tiempo de ejecución con elementos nombrados.

La forma más sencilla de crear una tupla literal es enumerar los valores deseados entre paréntesis. Esto crea una tupla con elementos sin nombre, a los que te puedes referir como Item1, Item2, y así sucesivamente:

```
var bob = ("Bob", 23);    // Allow compiler to infer the element types

Console.WriteLine (bob.Item1);   // Bob
Console.WriteLine (bob.Item2);   // 23
```

Las tuplas son tipos de valor con elementos mutables (lectura/escritura):

```
var joe = bob;                // joe is a *copy* of bob
joe.Item1 = "Joe";            // Change joe's Item1 from Bob to Joe
Console.WriteLine (bob);      // (Bob, 23)
Console.WriteLine (joe);      // (Joe, 23)
```

A diferencia de los tipos anónimos, puedes especificar explícitamente un tipo de tupla. Simplemente enumera cada uno de los tipos de elementos entre paréntesis:

```
(string,int) bob  = ("Bob", 23);
```

Esto significa que puedes devolver una tupla de forma útil desde un método:

```
(string,int) person = GetPerson();  // Could use 'var' instead if we want
Console.WriteLine (person.Item1);   // Bob
Console.WriteLine (person.Item2);   // 23

(string,int) GetPerson() => ("Bob", 23);
```

Las tuplas funcionan bien con los genéricos, por lo que los siguientes tipos son todos válidos:

```
Task<(string,int)>
Dictionary<(string,int),Uri>
IEnumerable<(int id, string name)>   // See below for naming elements
```

Nombrando los Elementos de las Tuplas

Opcionalmente, puedes dar nombres significativos a los elementos al crear tuplas literales:

```
var tuple = (name:"Bob", age:23);

Console.WriteLine (tuple.name);     // Bob
Console.WriteLine (tuple.age);      // 23
```

También puedes hacer lo mismo al especificar los tipos de las tuplas:

```
var person = GetPerson();
Console.WriteLine (person.name);    // Bob
Console.WriteLine (person.age);     // 23

(string name, int age) GetPerson() => ("Bob", 23);
```

Ten en cuenta que aún puedes tratar los elementos como anónimos y referirte a ellos como Item1, Item2, etc. (aunque Visual Studio oculta estos campos en IntelliSense).

Los nombres de los elementos se infieren automáticamente a partir de los nombres de propiedades o campos:

```
var now = DateTime.Now;
var tuple = (now.Day, now.Month, now.Year);
Console.WriteLine (tuple.Day);              // OK
```

Las tuplas son compatibles entre sí si sus tipos de elementos coinciden (en orden). Sus nombres de elementos no necesitan coincidir:

```
(string name, int age, char sex)  bob1 = ("Bob", 23, 'M');
(string age,  int sex, char name) bob2 = bob1;   // No error!
```

Nuestro ejemplo particular conduce a resultados confusos:

```
Console.WriteLine (bob2.name);    // M
Console.WriteLine (bob2.age);     // Bob
Console.WriteLine (bob2.sex);     // 23
```

Borrado de tipos (Type Erasure)

Mencionamos anteriormente que el compilador de C# maneja los tipos anónimos construyendo clases personalizadas con propiedades nombradas para cada uno de los elementos. Con las tuplas, C# funciona de manera diferente y utiliza una familia preexistente de estructuras genéricas:

```
public struct ValueTuple<T1>
public struct ValueTuple<T1,T2>
public struct ValueTuple<T1,T2,T3>
...
```

Cada una de las estructuras ValueTuple<> tiene campos llamados Item1, Item2, y así sucesivamente.

Por lo tanto, (string, int) es un alias de ValueTuple<string, int>, lo que significa que los elementos con nombre de la tupla no tienen nombres de propiedad correspondientes en los tipos subyacentes. En su lugar, los nombres existen solo en el código fuente y en la imaginación del compilador. En tiempo de ejecución, los nombres en su mayoría desaparecen, por lo que si descompilas un programa que hace referencia a elementos de tuplas nombrados, verás referencias solo a Item1, Item2, y así sucesivamente. Además, cuando examinas una variable de tupla en un depurador después de asignarla a un objeto (o la visualizas en LINQPad), los nombres de los elementos no están presentes. Y en su mayor parte, no puedes utilizar la reflexión (Capítulo 18) para determinar los nombres de los elementos de una tupla en tiempo de ejecución.



Dijimos que los nombres en su mayoría desaparecen porque hay una excepción. Con los métodos/propiedades que devuelven tipos de tuplas con nombres, el compilador emite los nombres de los elementos aplicando un atributo personalizado llamado TupleElementNamesAttribute (consultar "Atributos" en la página 227) al tipo de retorno del miembro. Esto permite que los elementos con nombre funcionen al llamar a métodos en un ensamblado diferente (para el cual el compilador no tiene el código fuente).

ValueTuple.Create

También puedes crear tuplas a través de un método de fábrica en el tipo (no genérico) ValueTuple:

```
ValueTuple<string,int> bob1 = ValueTuple.Create ("Bob", 23);
(string,int)           bob2 = ValueTuple.Create ("Bob", 23);
(string name, int age) bob3 = ValueTuple.Create ("Bob", 23);
```

Descomposición de Tuplas

Las tuplas admiten implícitamente el patrón de descomposición (ver "Desconstructores" en la página 20), por lo que puedes descomponer fácilmente una tupla en variables individuales. Considera lo siguiente:

```
var bob = ("Bob", 23);

string name = bob.Item1;
int age = bob.Item2;
```

Con el desconstructor de la tupla, puedes simplificar el código a esto:

```
var bob = ("Bob", 23);

(string name, int age) = bob;   // Deconstruct the bob tuple into
                                // separate variables (name and age).
Console.WriteLine (name);
Console.WriteLine (age);
```

La sintaxis para la descomposición es confusamente similar a la sintaxis para declarar una tupla con elementos nombrados. Lo siguiente resalta la diferencia:

```
(string name, int age)      = bob;   // Deconstructing a tuple
(string name, int age) bob2 = bob;   // Declaring a new tuple
```

Aquí tienes otro ejemplo, esta vez al llamar a un método y con inferencia de tipos (var):

```
var (name, age, sex) = GetBob();
Console.WriteLine (name);        // Bob
Console.WriteLine (age);         // 23
Console.WriteLine (sex);         // M

string, int, char) GetBob() => ( "Bob", 23, 'M');
```

También puedes deconstruir directamente en campos y propiedades, lo cual proporciona una forma sencilla de asignar valores a múltiples campos o propiedades en un constructor:

```
class Point
{
  public readonly int X, Y;
  public Point (int x, int y) => (X, Y) = (x, y);
}
```

Comparación de igualdad

Al igual que con los tipos anónimos, el método Equals realiza una comparación de igualdad estructural. Esto significa que compara los datos subyacentes en lugar de la referencia:

```
var t1 = ("one", 1);
var t2 = ("one", 1);
Console.WriteLine (t1.Equals (t2));    // True
```

Además, ValueTuple<> sobrecarga los operadores == y !=:

```
Console.WriteLine (t1 == t2);    // True (from C# 7.3)
```

Las tuplas también sobrescriben el método GetHashCode, lo que hace posible utilizar tuplas como claves en diccionarios. Cubrimos la comparación de igualdad en detalle en "Comparación de igualdad" en la página 324, y los diccionarios en el Capítulo 7.

Los tipos ValueTuple<> también implementan IComparable (consulte "Comparación de orden" en la página 335), lo que permite utilizar tuplas como clave de ordenamiento.

Las clases System.Tuple

También encontrarás otra familia de tipos genéricos en el espacio de nombres System llamados Tuple (en lugar de ValueTuple). Estos fueron introducidos en 2010 y se definieron como clases (mientras que los tipos ValueTuple son estructuras). En retrospectiva, se consideró un error definir las tuplas como clases: en los escenarios en los que las tuplas se utilizan comúnmente, las estructuras tienen una ligera ventaja de rendimiento (ya que evitan asignaciones de memoria innecesarias), con casi ninguna desventaja. Por lo tanto, cuando Microsoft agregó el soporte para tuplas en C# 7, se ignoraron los tipos Tuple existentes a favor de los nuevos ValueTuple. Aún puedes encontrar las clases Tuple en código escrito antes de C# 7. No tienen soporte especial del lenguaje y se utilizan de la siguiente manera:

```
Tuple<string,int> t = Tuple.Create ("Bob", 23);  // Factory method
Console.WriteLine (t.Item1);       // Bob
Console.WriteLine (t.Item2);       // 23
```

Registros (Records)

Un registro es un tipo especial de clase o estructura diseñado para funcionar bien con datos inmutables (de solo lectura). Su característica más útil es la mutación no destructiva; sin embargo, los registros también son útiles para crear tipos que simplemente combinan o contienen datos. En casos sencillos, eliminan el código repetitivo al tiempo que respetan la semántica de igualdad más adecuada para los tipos inmutables.

Los registros son puramente una construcción en tiempo de compilación de C#. En tiempo de ejecución, el CLR los trata simplemente como clases o estructuras (con un conjunto adicional de miembros "sintetizados" agregados por el compilador).

Antecedentes

Escribir tipos inmutables (cuyos campos no pueden modificarse después de la inicialización) es una estrategia popular para simplificar el software y reducir errores. También es un aspecto fundamental de la programación funcional, donde se evita el estado mutable y se tratan las funciones como datos. LINQ está inspirado en este principio.

Para "modificar" un objeto inmutable, debes crear uno nuevo y copiar los datos mientras incorporas las modificaciones (esto se llama mutación no destructiva). En cuanto al rendimiento, esto no es tan ineficiente como podrías esperar, porque siempre es suficiente hacer una copia superficial (una copia profunda, donde también se copian los subobjetos y colecciones, no es necesaria cuando los datos son inmutables). Pero en cuanto al esfuerzo de codificación, implementar la mutación no destructiva puede ser muy ineficiente, especialmente cuando hay muchas propiedades. Los registros resuelven este problema mediante un patrón compatible con el lenguaje.

Un segundo problema es que los programadores, especialmente los programadores funcionales, a veces utilizan tipos inmutables solo para combinar datos (sin agregar comportamiento). Definir

tales tipos requiere más trabajo del necesario, ya que se necesita un constructor para asignar cada parámetro a cada propiedad (también puede ser útil un desconstructor). Con los registros, el compilador puede hacer este trabajo por ti.

Finalmente, una de las consecuencias de que un objeto sea inmutable es que su identidad no puede cambiar, lo que significa que es más útil que estos tipos implementen igualdad estructural en lugar de igualdad referencial. La igualdad estructural significa que dos instancias son iguales si sus datos son iguales (como en las tuplas). Los registros te brindan igualdad estructural de forma predeterminada, sin importar si el tipo subyacente es una clase o una estructura, sin ningún código repetitivo.

Definición de un registro

Una definición de registro es similar a una definición de clase o estructura y puede contener los mismos tipos de miembros, incluyendo campos, propiedades, métodos, y más. Los registros pueden implementar interfaces y los registros basados en clases pueden heredar de otros registros basados en clases.

De forma predeterminada, el tipo subyacente de un registro es una clase:

```
record Point { }          // Point is a class
```

A partir de C# 10, el tipo subyacente de un registro también puede ser una estructura:

```
 record struct Point { }   // Point is a struct
```

(También es válido `record class` y tiene el mismo significado que `record`).

Un registro simple podría contener solo un conjunto de propiedades de solo lectura (`init-only`) y tal vez un constructor:

```
record Point
{
  public Point (double x, double y) => (X, Y) = (x, y);

  public double X { get; init; }
  public double Y { get; init; }
}
```



Nuestro constructor utiliza un atajo que describimos en la sección anterior.

```
(X, Y) = (x, y);
```

es equivalente (en este caso) a lo siguiente:

```
{ this.X = x; this.Y = y; }
```

Al compilar, C# transforma la definición del registro en una clase (o estructura) y realiza los siguientes pasos adicionales:

- Escribe un constructor de copia protegido (y un método Clone oculto) para facilitar la mutación no destructiva.
- Sobrescribe/sobrecarga las funciones relacionadas con la igualdad para implementar la igualdad estructural.
- Sobrescribe el método ToString() para expandir las propiedades públicas del registro, al igual que en los tipos anónimos.

La declaración de registro anterior se expande aproximadamente a algo como esto:

```
class Point
{
  public Point (double x, double y) => (X, Y) = (x, y);

  public double X { get; init; }
  public double Y { get; init; }

  protected Point (Point original)    // “Copy constructor”
  {
    this.X = original.X; this.Y = original.Y
  }

  // This method has a strange compiler-generated name:
  public virtual Point <Clone>$() => new Point (this);   // Clone method

  // Additional code to override Equals, ==, !=, GetHashCode, ToString()
  // ...
}
```



Si bien no hay nada que te impida poner parámetros opcionales en el constructor, un buen patrón (al menos en bibliotecas públicas) es omitirlos en el constructor y exponerlos únicamente como propiedades de solo lectura (`init-only`):

```
new Foo (123, 234) { Optional2 = 345 };

record Foo
{
  public Foo (int required1, int required2) { ... }

  public int Required1 { get; init; }
  public int Required2 { get; init; }

  public int Optional1 { get; init; }
  public int Optional2 { get; init; }
}
```

La ventaja de este patrón es que puedes agregar de forma segura propiedades `init-only` más adelante sin romper la compatibilidad binaria con los consumidores que han compilado contra versiones anteriores de tu ensamblado.

Listas de parámetros

Una definición de registro también puede incluir una lista de parámetros:

```
record Point (double X, double Y)
{
  // You can optionally define additional class members here...
}
```

Si se especifica una lista de parámetros, el compilador realiza los siguientes pasos adicionales:

- Escribe una propiedad `init-only` por parámetro.
- Escribe un constructor principal para inicializar las propiedades.
- Escribe un deconstructor.

La mutabilidad con las estructuras de registro

Cuando defines una lista de parámetros en una estructura de registro, el compilador emite propiedades modificables en lugar de propiedades de solo lectura, a menos que prefijes la declaración del registro con `readonly`:

```
readonly record struct Point (double X, double Y);
```

La razón de esto es que en casos de uso típicos, los beneficios de seguridad de la inmutabilidad surgen no de una estructura siendo inmutable, sino de su contenedor siendo inmutable. En el siguiente ejemplo, no podemos modificar el campo `X`, aunque `X` sea modificable:

```
var test = new Immutable();
test.Field.X++;  // Prohibited, because Field is readonly
test.Prop.X++;   // Prohibited, because Prop is {get;} only

class Immutable
{
  public readonly Mutable Field;
  public Mutable Prop { get; }
}

struct Mutable { public int X, Y; }
```

Y aunque podríamos hacer lo siguiente:

```
var test = new Immutable();
Mutable m = test.Prop;
m.X++;
```

todo lo que lograríamos es modificar una variable local (una copia de `test.Prop`). La mutación de una variable local puede ser una optimización útil y no invalida los beneficios de un sistema de tipos inmutable.

Por otro lado, si hicimos que `Field` sea un campo modificable y `Prop` una propiedad modificable, simplemente podríamos reemplazar su contenido, independientemente de cómo se haya declarado la estructura `Mutable`.

Esto significa que si declaramos nuestro registro Point simplemente como:

```
record Point (double X, double Y);
```

el compilador generará (casi) exactamente lo que enumeramos en la expansión anterior. Una pequeña diferencia es que los nombres de los parámetros en el constructor principal terminarán siendo X e Y en lugar de x e y:

```
public Point (double X, double Y)   // “Primary constructor”
{
  this.X = X; this.Y = Y;
}
```



Además, debido a ser un constructor principal, los parámetros X e Y estarán mágicamente disponibles para cualquier inicializador de campo o propiedad en tu registro. Discutimos las sutilezas de esto más adelante, en "Constructores principales" en la página 219.

Otra diferencia, cuando defines una lista de parámetros, es que el compilador también genera un deconstructor:

```
public void Deconstruct (out double X, out double Y)   // Deconstructor
{
  X = this.X; Y = this.Y;
}
```

Los registros con listas de parámetros pueden ser subclaseados utilizando la siguiente sintaxis:

```
record Point3D (double X, double Y, double Z) : Point (X, Y);
```

El compilador luego emite un constructor principal de la siguiente manera:

```
class Point3D : Point
{
  public double Z { get; init; }

  public Point3D (double X, double Y, double Z) : base (X, Y)
    => this.Z = Z;
}
```

v



Las listas de parámetros ofrecen una forma conveniente cuando necesitas una clase que simplemente agrupa un conjunto de valores (un tipo de producto en la programación

funcional) y también pueden ser útiles para prototipado. Como veremos más adelante, no son tan útiles cuando necesitas agregar lógica a los accesores init (como validación de argumentos).

Mutación no destructiva

El paso más importante que realiza el compilador con todos los registros es escribir un constructor de copia (y un método Clone oculto). Esto permite la mutación no destructiva mediante la palabra clave "with":

```
Point p1 = new Point (3, 3);
Point p2 = p1 with { Y = 4 };
Console.WriteLine (p2);       // Point { X = 3, Y = 4 }

record Point (double X, double Y);
```

En este ejemplo, p2 es una copia de p1, pero con su propiedad Y establecida en 4. El beneficio es más evidente cuando hay más propiedades:

```
Test t1 = new Test (1, 2, 3, 4, 5, 6, 7, 8);
Test t2 = t1 with { A = 10, C = 30 };
Console.WriteLine (t2);

record Test (int A, int B, int C, int D, int E, int F, int G, int H);
```

Aquí está la salida:

```
Test { A = 10, B = 2, C = 30, D = 4, E = 5, F = 6, G = 7, H = 8 }
```

La mutación no destructiva ocurre en dos fases:

1. Primero, el constructor de copia clona el registro. De forma predeterminada, copia cada uno de los campos subyacentes del registro, creando una réplica fiel evitando cualquier lógica en los accesores init. Se incluyen todos los campos (públicos y privados, así como los campos ocultos que respaldan las propiedades automáticas).

2. Luego, cada propiedad en la lista de inicialización de miembros se actualiza (esta vez utilizando los accesores init).

El compilador traduce el código:

```
Test t2 = t1 with { A = 10, C = 30 };
```

en algo funcionalmente equivalente a lo siguiente:

```
Test t2 = new Test(t1);  // Use copy constructor to clone t1 field by field
t2.A = 10;               // Update property A
t2.C = 30;               // Update property C
```

(El mismo código no compilaría si lo escribieras explícitamente porque A y C

son propiedades de solo inicialización. Además, el constructor de copia está protegido; C# lo resuelve invocándolo a través de un método oculto público que escribe en el registro llamado <Clone>$.)

Si es necesario, puedes definir tu propio constructor de copia. C# utilizará tu definición en lugar de generar uno automáticamente:

```
protected Point (Point original)
{
  this.X = original.X; this.Y = original.Y;
}
```

Escribir un constructor de copia personalizado puede ser útil si tu registro contiene subobjetos o colecciones mutables que deseas clonar, o si hay campos calculados que deseas borrar. Desafortunadamente, solo puedes reemplazar, no mejorar, la implementación predeterminada.

 Al heredar de otro registro, el constructor de copia es responsable de copiar solo sus propios campos. Para copiar los campos del registro base, debes delegar en la base:

```
protected Point (Point original) : base (original)
{
  ...
}
```

Validación de Propiedades

Con propiedades explícitas, puedes agregar lógica de validación en los accesores init. En

el siguiente ejemplo, nos aseguramos de que X nunca pueda ser NaN (Not a Number):

```
record Point
{
  // Notice that we assign x to the X property (and not the _x field):
  public Point (double x, double y) => (X, Y) = (x, y);

  double _x;
  public double X
  {
    get => _x;
    init
    {
      if (double.IsNaN (value))
        throw new ArgumentException ("X Cannot be NaN");
      _x = value;
    }
  }

  public double Y { get; init; }
}
```

Nuestro diseño asegura que la validación ocurra tanto durante la construcción como cuando el objeto sufre mutaciones no destructivas:

```
Point p1 = new Point (2, 3);
Point p2 = p1 with { X = double.NaN };   // throws an exception
```

Recuerda que el constructor de copia generado automáticamente copia todos los campos y propiedades automáticas. Esto significa que el constructor de copia generado ahora se verá así:

```
protected Point (Point original)
 {
   _x = original._x; Y = original.Y;
 }
```

Observa que la copia del campo _x evade el accesor de la propiedad X. Sin embargo, esto no puede romper nada, porque está copiando fielmente un objeto que ya ha sido poblado de manera segura a través del accesor init de X.

Campos Calculados y Evaluación Perezosa

Un patrón popular de programación funcional que funciona bien con tipos inmutables es la evaluación perezosa, donde un valor no se calcula hasta que es requerido y luego se almacena en caché para su reutilización. Supongamos, por ejemplo, que queremos definir una propiedad en nuestro registro Point que devuelva la distancia desde el origen (0, 0):

```
record Point (double X, double Y)
{
  public double DistanceFromOrigin => Math.Sqrt (X*X + Y*Y);
}
```

Ahora intentemos refactorizar esto para evitar el costo de recalcular DistanceFromOrigin cada vez que se accede a la propiedad. Comenzaremos eliminando la lista de propiedades y definiendo X, Y y DistanceFromOrigin como propiedades de solo lectura. Luego podemos calcular este último en el constructor:

```
record Point
{
  public double X { get; }
  public double Y { get; }
  public double DistanceFromOrigin { get; }

  public Point (double x, double y) =>
    (X, Y, DistanceFromOrigin) = (x, y, Math.Sqrt (x*x + y*y));
}
```

Esto funciona, pero no permite la mutación no destructiva (cambiar X e Y a propiedades de solo lectura inicial no sería válido porque DistanceFromOrigin se volvería obsoleto después de que se ejecuten los accesor init). También es subóptimo en el sentido de que el cálculo siempre se realiza, independientemente de si se lee alguna vez la propiedad DistanceFromOrigin. La

solución óptima es almacenar en caché su valor en un campo y poblarlo de forma perezosa (en la primera utilización):

```
record Point
{
  ...

  double? _distance;
  public double DistanceFromOrigin
  {
    get
    {
      if (_distance == null)
        _distance = Math.Sqrt (X*X + Y*Y);

      return _distance.Value;
    }
  }
}
```



Técnicamente, mutamos _distance en este código. Sin embargo, sigue siendo válido llamar a Point un tipo inmutable. Mutar un campo únicamente para poblar un valor perezoso no invalida los principios o beneficios de la inmutabilidad, e incluso puede ocultarse mediante el uso del tipo Lazy<T> que describimos en el Capítulo 21.

Con el operador de asignación null-coalescing (??=) de C#, podemos reducir toda la declaración de la propiedad a una línea de código:

```
public double DistanceFromOrigin => _distance ??= Math.Sqrt (X*X + Y*Y);
```

(Esto significa: devuelve _distance si no es nulo; de lo contrario, devuelve Math.Sqrt(X*X + Y*Y) mientras se le asigna a _distance).

Para que esto funcione con propiedades de solo lectura inicial, necesitamos dar un paso adicional, que es limpiar el campo _distance en caché cuando se actualiza X o Y a través del accesor init. Aquí está el código completo:

```
record Point
{
  public Point (double x, double y) => (X, Y) = (x, y);

  double _x, _y;
  public double X { get => _x; init { _x = value; _distance = null; } }
  public double Y { get => _y; init { _y = value; _distance = null; } }

  double? _distance;
  public double DistanceFromOrigin => _distance ??= Math.Sqrt (X*X + Y*Y);
}
```

Ahora se puede realizar mutación no destructiva en Point:

```
 Point p1 = new Point (2, 3);
 Console.WriteLine (p1.DistanceFromOrigin);   // 3.605551275463989

Point p2 = p1 with { Y = 4 };
Console.WriteLine (p2.DistanceFromOrigin);   // 4.47213595499958
```

Una ventaja adicional es que el constructor de copia autogenerado copia el campo en caché _distance. Esto significa que si un registro tiene otras propiedades que no están involucradas en el cálculo, una mutación no destructiva de esas propiedades no desencadenaría una pérdida innecesaria del valor en caché. Si no te importa esta ventaja, una alternativa para limpiar el valor en caché en los accesores init es escribir un constructor de copia personalizado que ignore el campo en caché. Esto es más conciso porque funciona con listas de parámetros y el constructor de copia personalizado puede aprovechar el deconstructor:

```
record Point (double X, double Y)
{
  double? _distance;
  public double DistanceFromOrigin => _distance ??= Math.Sqrt (X*X + Y*Y);

  protected Point (Point other) => (X, Y) = other;
}
```

Ten en cuenta que con cualquiera de estas soluciones, la adición de campos calculados de forma perezosa rompe la comparación de igualdad estructural predeterminada (porque dichos campos pueden estar poblados o no), aunque veremos pronto que es relativamente fácil solucionarlo.

Constructores Primarios

Cuando defines un registro con una lista de parámetros, el compilador genera automáticamente las declaraciones de propiedades, así como un constructor primario (y un deconstructor). Como hemos visto, esto funciona bien en casos simples, y en casos más complejos puedes omitir la lista de parámetros y escribir las declaraciones de propiedades y el constructor manualmente.

C# también ofrece una opción intermedia ligeramente útil, si estás dispuesto a lidiar con las curiosas semánticas de los constructores primarios, que es definir una lista de parámetros mientras escribes algunas o todas las declaraciones de propiedades tú mismo:

```
record Student (string ID, string LastName, string GivenName)
{
  public string ID { get; } = ID;
}
```

En este caso, "tomamos el control" de la definición de la propiedad ID, definiéndola como de solo lectura (en lugar de inicialización solo de lectura), evitando que participe en la mutación no destructiva. Si nunca necesitas mutar de manera no destructiva una propiedad en particular, hacerla de solo lectura te permite almacenar datos calculados en el registro sin tener que codificar un mecanismo de actualización.

Observa que necesitamos incluir un inicializador de propiedad (en negrita):

```
public string ID { get; } = ID;
```

Cuando "tomas el control" de una declaración de propiedad, te haces responsable de inicializar su valor; el constructor primario ya no lo hace automáticamente. Ten en cuenta que el ID en negrita se refiere al parámetro del constructor primario, no a la propiedad ID.



Con las estructuras de registros, es legal redefinir una propiedad como un campo:

```
record struct Student (string ID)
{
  public string ID = ID;
}
```

Una característica única de los constructores primarios es que sus parámetros (ID, LastName y FirstName en este caso) son mágicamente visibles para todos los inicializadores de campos y propiedades. Podemos ilustrar esto ampliando nuestro ejemplo de la siguiente manera:

```
record Student (string ID, string LastName, string FirstName)
{
  public string ID { get; } = ID;
  readonly int _enrollmentYear = int.Parse (ID.Substring (0, 4));
}
```

Nuevamente, el ID en negrita se refiere al parámetro del constructor primario, no a la propiedad (la razón por la que no hay ambigüedad es que es ilegal acceder a propiedades desde inicializadores). En este ejemplo, calculamos _enrollmentYear a partir de los primeros cuatro dígitos de ID. Si bien es seguro almacenar esto en un campo de solo lectura (porque la propiedad ID es de solo lectura y, por lo tanto, no se puede mutar de manera no destructiva), este código no funcionaría tan bien en el mundo real. Esto se debe a que sin un constructor explícito, no hay un lugar central en el que validar ID y lanzar una excepción significativa en caso de que sea inválido (un requisito común).

La validación también es una buena razón para necesitar escribir accesores init solo (como discutimos en "Validación de propiedades" en la página 216). Desafortunadamente, los constructores primarios no funcionan bien en este escenario. Para ilustrarlo, considera el siguiente registro, donde un accesor init realiza una verificación de nulidad:

```
record Person (string Name)
{
  string _name = Name;
  public string Name
  {
    get  => _name;
    init => _name = value ?? throw new ArgumentNullException ("Name");
  }
}
```

Como Name no es una propiedad automática, no puede definir un inicializador. Lo mejor que podemos hacer es poner el inicializador en el campo de respaldo (en negrita). Desafortunadamente, al hacerlo se omite la verificación de nulidad:

```
var p = new Person (null);     // Succeeds! (bypasses the null check)
```

La dificultad radica en que no hay forma de asignar un parámetro del constructor primario a una propiedad sin escribir el constructor nosotros mismos. Aunque existen soluciones alternativas (como factorizar la lógica de validación init en un método estático separado que llamamos dos veces), la solución más simple es evitar por completo la lista de parámetros y escribir manualmente un constructor ordinario (y, si es necesario, un deconstructor):

```
record Person
{
  public Person (string name) => Name = name;  // Assign to *PROPERTY*

  string _name;
  public string Name { get => _name; init => ... }
}
```

Registros y comparación de igualdad

Al igual que con las estructuras, los tipos anónimos y las tuplas, los registros proporcionan igualdad estructural de forma predeterminada, lo que significa que dos registros son iguales si sus campos (y propiedades automáticas) son iguales:

```
var p1 = new Point (1, 2);
var p2 = new Point (1, 2);
Console.WriteLine (p1.Equals (p2));   // True

record Point (double X, double Y);
```

El operador de igualdad también funciona con registros (al igual que con tuplas):

```
Console.WriteLine (p1 == p2);         // True
```

La implementación de igualdad predeterminada para registros es inevitablemente frágil. En particular, se rompe si el registro contiene valores perezosos, valores transitorios, matrices o tipos de colección (que requieren un manejo especial para la comparación de igualdad).

Afortunadamente, es relativamente fácil solucionarlo (si necesitas que la igualdad funcione), y hacerlo requiere menos trabajo que agregar un comportamiento completo de igualdad a clases o estructuras.

A diferencia de las clases y estructuras, no puedes anular (override) el método object.Equals; en su lugar, debes definir un método Equals público con la siguiente firma:

```
record Point (double X, double Y)
{
  double _someOtherField;
  public virtual bool Equals (Point other) =>
    other != null && X == other.X && Y == other.Y;
}
```

El método Equals debe ser virtual (no override) y debe ser fuertemente tipado de manera que acepte el tipo de registro real (Point en este caso, no object). Una vez que tengas la firma correcta, el compilador parcheará automáticamente tu método.

En nuestro ejemplo, cambiamos la lógica de igualdad para que solo compare X e Y (ignorando _someOtherField).

Si heredas de otro registro, puedes llamar al método base.Equals:

```
public virtual bool Equals (Point other) => base.Equals (other) && ...
```

Como cualquier tipo, si tomas el control de la comparación de igualdad, también debes anular el método GetHashCode(). Una característica interesante de los registros es que no sobrecargas los operadores != o ==, ni implementas IEquatable<T>: todo esto se hace por ti. Cubrimos este tema de la comparación de igualdad completamente en "Comparación de igualdad" en la página 324.

Patrones

En el Capítulo 3, demostramos cómo usar el operador is para probar si una conversión de referencia tendrá éxito:

```
if (obj is string)
  Console.WriteLine (((string)obj).Length);
```

O de forma más concisa:

```
if (obj is string s)
  Console.WriteLine (s.Length);
```

Esto emplea un tipo de patrón llamado patrón de tipo. El operador is también admite otros patrones que se introdujeron en versiones recientes de C#, como el patrón de propiedad:

```
if (obj is string { Length:4 })
  Console.WriteLine ("A string with 4 characters");
```

Los patrones se admiten en los siguientes contextos:

- Después del operador is (variable is patrón)
- En declaraciones switch
- En expresiones switch

Ya hemos cubierto el patrón de tipo (y brevemente, el patrón de tupla) en "Cambiar entre tipos" en la página 81 y "El operador is" en la página 118. En esta sección, cubrimos patrones más avanzados que se introdujeron en versiones recientes de C#.

Algunos de los patrones más especializados están destinados a ser utilizados en declaraciones/expressions switch. Aquí, reducen la necesidad de cláusulas when y te permiten usar switch donde antes no era posible.



Los patrones en esta sección son útiles en algunos escenarios, pero no en todos. Recuerda que siempre puedes reemplazar expresiones switch altamente patrocinadas con simples instrucciones if, o en algunos casos, con el operador condicional ternario, a menudo sin mucho código adicional.

Patrón var

El patrón var es una variación del patrón de tipo en el que reemplazas el nombre del tipo con la palabra clave var. La conversión siempre tiene éxito, por lo que su propósito es simplemente permitirte reutilizar la variable que sigue al patrón:

```
bool IsJanetOrJohn (string name) =>
  name.ToUpper() is var upper && (upper == "JANET" || upper == "JOHN");
```

Esto es equivalente a:

```
bool IsJanetOrJohn (string name)
{
  string upper = name.ToUpper();
  return upper == "JANET" || upper == "JOHN";
}
```

La capacidad de introducir y reutilizar una variable intermedia (en este caso, `upper`) en un método expresado en cuerpo de expresión es conveniente. Desafortunadamente, tiende a ser útil solo cuando el método en cuestión tiene un tipo de retorno bool.

Patrón Constante(Constant Pattern)

El patrón constante te permite hacer coincidir directamente con una constante y es útil cuando trabajas con el tipo object:

```
void Foo (object obj)
{
  if (obj is 3) ...
}
```

Esta expresión en negrita es equivalente a lo siguiente:

```
obj is int && (int)obj == 3
```

(Siendo un operador estático, C# no permite usar == para comparar directamente un objeto con una constante, porque el compilador necesita conocer los tipos de antemano).

Por sí solo, este patrón es solo marginalmente útil, ya que hay una alternativa razonable:

```
if (3.Equals (obj)) ...
```

Como veremos pronto, el patrón constante puede volverse más útil con los combinadores de patrones.

Patrones Relacionales (Relational Patterns)

A partir de C# 9, puedes usar los operadores <, >, <= y >= en los patrones:

```
if (x is > 100) Console.WriteLine ("x is greater than 100");
```

Esto se vuelve significativamente útil en un switch:

```
string GetWeightCategory (decimal bmi) => bmi switch
{
  < 18.5m => "underweight",
  < 25m => "normal",
  < 30m => "overweight",
  _ => "obese"
};
```

Los patrones relacionales se vuelven aún más útiles en conjunción con los combinadores de patrones.



El patrón relacional también funciona cuando la variable tiene un tipo de objeto en tiempo de compilación, pero debes tener mucho cuidado con el uso de constantes numéricas. En el siguiente ejemplo, la última línea imprime False porque estamos intentando hacer coincidir un valor decimal con un literal entero:

```
object obj = 2m;                 // obj is decimal
Console.WriteLine (obj is < 3m);  // True
Console.WriteLine (obj is < 3);   // False
```

Combinadores de patrones

A partir de C# 9, puedes usar las palabras clave "and", "or" y "not" para combinar patrones:

```
bool IsJanetOrJohn (string name) => name.ToUpper() is "JANET" or "JOHN";

bool IsVowel (char c) => c is 'a' or 'e' or 'i' or 'o' or 'u';

bool Between1And9 (int n) => n is >= 1 and <= 9;

bool IsLetter (char c) => c is >= 'a' and <= 'z'
                          or >= 'A' and <= 'Z';
```

Al igual que los operadores && y ||, "and" tiene una precedencia mayor que "or". Puedes anular esto utilizando paréntesis.

Un buen truco es combinar el combinador "not" con el patrón de tipo para comprobar si un objeto es (no) de un tipo:

```
if (obj is not string) ...
```

Esto se ve más bonito que:

```
if (!(obj is string)) ...
```

Patrones de tupla y posicionales

El patrón de tupla (introducido en C# 8) coincide con tuplas:

```
var p = (2, 3);
Console.WriteLine (p is (2, 3));   // True
```

Puedes usar esto para hacer conmutación en múltiples valores:

```
int AverageCelsiusTemperature (Season season, bool daytime) =>
  (season, daytime) switch
  {
    (Season.Spring, true) => 20,
    (Season.Spring, false) => 16,
    (Season.Summer, true) => 27,
    (Season.Summer, false) => 22,
    (Season.Fall, true) => 18,
    (Season.Fall, false) => 12,
    (Season.Winter, true) => 10,

    (Season.Winter, false) => -2,
    _ => throw new Exception ("Unexpected combination")
};

enum Season { Spring, Summer, Fall, Winter };
```

El patrón de tupla se puede considerar un caso especial del patrón posicional (C# 8+), que coincide con cualquier tipo que exponga un método Deconstruct (ver "Deconstructores" en la página 20). En el siguiente ejemplo, aprovechamos el deconstructor generado por el compilador del registro Point:

```
var p = new Point (2, 2);
Console.WriteLine (p is (2, 2));  // True

record Point (int X, int Y);      // Has compiler-generated deconstructor
```

Puedes deconstruir mientras haces coincidir utilizando la siguiente sintaxis:

```
Console.WriteLine (p is (var x, var y) && x == y);   // True
```

Aquí hay una expresión switch que combina un patrón de tipo con un patrón posicional:

```
string Print (object obj) => obj switch
{
  Point (0, 0)                      => "Empty point",
  Point (var x, var y) when x == y  => "Diagonal"
  ...
};
```

Patrones de propiedades

Un patrón de propiedad (C# 8+) coincide con uno o más valores de propiedad de un objeto. Damos un ejemplo simple anteriormente en el contexto del operador is:

```
if (obj is string { Length:4 }) ...
```

Sin embargo, esto no ahorra mucho en comparación con lo siguiente:

```
if (obj is string s && s.Length == 4) ...
```

Con las sentencias y expresiones switch, los patrones de propiedades son más útiles. Considera la clase System.Uri, que representa una URI. Tiene propiedades que incluyen Scheme, Host, Port y IsLoopback. Al escribir un firewall, podríamos decidir si permitir o bloquear una URI utilizando una expresión switch que utilice patrones de propiedades:

```
bool ShouldAllow (Uri uri) => uri switch
{
  { Scheme: "http",  Port: 80  } => true,
  { Scheme: "https", Port: 443 } => true,
  { Scheme: "ftp",   Port: 21  } => true,
  { IsLoopback: true           } => true,
  _ => false
};
```

Puedes anidar propiedades, lo que hace que la siguiente cláusula sea válida:

```
{ Scheme: { Length: 4 }, Port: 80 } => true,
```

que, a partir de C# 10, se puede simplificar a:

```
{ Scheme.Length: 4, Port: 80 } => true,
```

Puedes usar otros patrones dentro de los patrones de propiedades, incluyendo el patrón relacional:

```
{ Host: { Length: < 1000 }, Port: > 0 } => true,
```

Condiciones más elaboradas se pueden expresar con una cláusula when:

```
{ Scheme: "http" } when string.IsNullOrWhiteSpace (uri.Query) => true,
```

También puedes combinar el patrón de propiedad con el patrón de tipo:

```
bool ShouldAllow (object uri) => uri switch
{
  Uri { Scheme: "http",  Port: 80  } => true,
  Uri { Scheme: "https", Port: 443 } => true,
  ...
```

Como puedes esperar con los patrones de tipo, puedes introducir una variable al final de una cláusula y luego utilizar esa variable:

```
Uri { Scheme: "http", Port: 80 } httpUri => httpUri.Host.Length < 1000,
```

También puedes usar esa variable en una cláusula when:

```
  Uri { Scheme: "http", Port: 80 } httpUri
                                   when httpUri.Host.Length < 1000 => true,
```

Una característica algo extraña de los patrones de propiedades es que también puedes introducir variables a nivel de propiedad:

```
{ Scheme: "http", Port: 80, Host: string host } => host.Length < 1000,
```

Se permite la inferencia de tipo implícita, por lo que puedes sustituir "string" por "var". Aquí tienes un ejemplo completo:

```
bool ShouldAllow (Uri uri) => uri switch
{
  { Scheme: "http",  Port: 80, Host: var host } => host.Length < 1000,
  { Scheme: "https", Port: 443               } => true,
  { Scheme: "ftp",   Port: 21                } => true,
  { IsLoopback: true                         } => true,
  _ => false
};
```

Es difícil inventar ejemplos en los que esto ahorre más que unos pocos caracteres. En nuestro caso, la alternativa en realidad es más corta:

```
{ Scheme: "http", Port: 80 } => uri.Host.Length < 1000 => ...
```

O:

```
 { Scheme: "http", Port: 80, Host: { Length: < 1000 } } => ...
```

Atributos

Ya estás familiarizado con la noción de atribuir elementos de código de un programa con modificadores, como virtual o ref. Estos constructos están incorporados en el lenguaje. Los atributos son un mecanismo extensible para agregar información personalizada a los elementos de código (ensamblados, tipos, miembros, valores de retorno, parámetros y parámetros de tipo genérico). Esta capacidad de extensión es útil para servicios que se integran profundamente en el sistema de tipos, sin requerir palabras clave o constructos especiales en el lenguaje C#.

Un escenario adecuado para los atributos es la serialización, que es el proceso de convertir objetos arbitrarios a un formato específico para su almacenamiento o transmisión. En este escenario, un atributo en un campo puede especificar la traducción entre la representación del campo en C# y la representación del campo en el formato de datos.

Clases de atributos

Un atributo se define mediante una clase que hereda (directa o indirectamente) de la clase abstracta System.Attribute. Para adjuntar un atributo a un elemento de código, se especifica el nombre del tipo de atributo entre corchetes, antes del elemento de código. Por ejemplo, lo siguiente adjunta el atributo ObsoleteAttribute a la clase Foo:

```
[ObsoleteAttribute]
public class Foo {...}
```

Este atributo en particular es reconocido por el compilador y generará advertencias del compilador si se hace referencia a un tipo o miembro marcado como obsoleto. Por convención, todos los tipos de atributos terminan con la palabra "Attribute". C# reconoce esto y te permite omitir el sufijo al adjuntar un atributo:

```
[Obsolete]
public class Foo {...}
```

ObsoleteAttribute es un tipo declarado en el espacio de nombres System como sigue (simplificado por brevedad):

```
public sealed class ObsoleteAttribute : Attribute {...}
```

Las bibliotecas .NET incluyen muchos atributos predefinidos. Describiremos cómo escribir tus propios atributos en el Capítulo 18.

Parámetros de atributos con nombre y posicionales

Los atributos pueden tener parámetros. En el siguiente ejemplo, aplicamos el atributo XmlTypeAttribute a una clase. Este atributo instruye al serializador XML (en System.Xml.Serialization) sobre cómo se representa un objeto en XML y acepta varios parámetros de atributo. El siguiente atributo asigna la clase CustomerEntity a un elemento XML llamado Customer, que pertenece al espacio de nombres http://oreilly.com:

```
[XmlType ("Customer", Namespace="http://oreilly.com")]
public class CustomerEntity { ... }
```

Los parámetros de los atributos se dividen en dos categorías: posicionales o con nombre. En el ejemplo anterior, el primer argumento es un parámetro posicional, mientras que el segundo es un parámetro con nombre. Los parámetros posicionales corresponden a los parámetros de los constructores públicos del tipo de atributo. Los parámetros con nombre corresponden a campos públicos o propiedades públicas en el tipo de atributo.

Al especificar un atributo, debes incluir los parámetros posicionales que correspondan a uno de los constructores del atributo. Los parámetros con nombre son opcionales. En el Capítulo 18 se describen los tipos de parámetros válidos y las reglas para su evaluación.

Aplicar atributos a ensamblados y campos respaldados

Implícitamente, el objetivo de un atributo es el elemento de código que le precede inmediatamente, que suele ser un tipo o miembro de tipo. Sin embargo, también puedes adjuntar atributos a un ensamblado. Esto requiere que especifiques explícitamente el objetivo del atributo. Aquí te mostramos cómo puedes usar el atributo AssemblyFileVersion para adjuntar una versión al ensamblado:

```
[assembly: AssemblyFileVersion ("1.2.3.4")]
```

A partir de C# 7.3, puedes utilizar el prefijo field: para aplicar un atributo a los campos respaldados de una propiedad automática. Esto puede ser útil para controlar la serialización:

```
[field:NonSerialized]
public int MyProperty { get; set; }
```

Aplicar atributos a expresiones lambda (C# 10)

A partir de C# 10, puedes aplicar atributos al método, los parámetros y el valor de retorno de una expresión lambda:

```
Action<int> a = [Description ("Method")]
                [return: Description ("Return value")]
                ([Description ("Parameter")]int x) => Console.Write (x);
```



Esto es útil cuando trabajas con frameworks, como ASP.NET, que dependen de que coloques atributos en los métodos que escribes. Con esta característica, puedes evitar tener que crear métodos con nombre para operaciones simples.

Estos atributos se aplican al método generado por el compilador al que apunta el delegado. En el Capítulo 18, describiremos cómo reflejar sobre los atributos en el código. Por ahora, aquí tienes el código adicional que necesitas para resolver esa indirección:

```
var methodAtt = a.GetMethodInfo().GetCustomAttributes();
var paramAtt = a.GetMethodInfo().GetParameters()[0].GetCustomAttributes();
var returnAtt = a.GetMethodInfo().ReturnParameter.GetCustomAttributes();
```

Para evitar ambigüedades sintácticas al aplicar atributos a un parámetro en una expresión lambda, siempre se requieren paréntesis. No se permiten atributos en expresiones lambda de árbol de expresión.

Especificación de múltiples atributos

Puedes especificar varios atributos para un solo elemento de código. Puedes enumerar cada atributo dentro del mismo par de corchetes (separados por comas) o en pares de corchetes separados (o una combinación de ambos). Los siguientes tres ejemplos son semánticamente idénticos:

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar {...}

[Serializable] [Obsolete] [CLSCompliant(false)]
public class Bar {...}

[Serializable, Obsolete]
[CLSCompliant(false)]
public class Bar {...}
```

Atributos Caller Info

Puedes etiquetar los parámetros opcionales con uno de los tres atributos de Caller Info, que instruyen al compilador a proporcionar información obtenida del código fuente del llamador en el valor predeterminado del parámetro:

- [CallerMemberName] aplica el nombre del miembro del llamador.
- CallerFilePath] aplica la ruta al archivo de código fuente del llamador.
- [CallerLineNumber] aplica el número de línea en el archivo de código fuente del llamador.

El método Foo en el siguiente programa demuestra los tres:

```
using System;
using System.Runtime.CompilerServices;

class Program
{
  static void Main() => Foo();

  static void Foo (
    [CallerMemberName] string memberName = null,
    [CallerFilePath] string filePath = null,
    [CallerLineNumber] int lineNumber = 0)
  {
    Console.WriteLine (memberName);
    Console.WriteLine (filePath);
    Console.WriteLine (lineNumber);
  }
}
```

Suponiendo que nuestro programa se encuentra en c:\source\test\Program.cs, la salida sería:

```
Main
c:\source\test\Program.cs
6
```

Al igual que con los parámetros opcionales estándar, la sustitución se realiza en el sitio de llamada. Por lo tanto, nuestro método Main es azúcar sintáctico para esto:

```
static void Main() => Foo ("Main", @"c:\source\test\Program.cs", 6);
```

Los atributos Caller Info son útiles para el registro (logging) y para implementar patrones como la activación de un único evento de notificación de cambios cuando cambia cualquier propiedad de un objeto. De hecho, hay una interfaz estándar para esto en el espacio de nombres System.ComponentModel, llamada INotifyPropertyChanged:

```
public interface INotifyPropertyChanged
{
  event PropertyChangedEventHandler PropertyChanged;
}

public delegate void PropertyChangedEventHandler
  (object sender, PropertyChangedEventArgs e);

public class PropertyChangedEventArgs : EventArgs
{
  public PropertyChangedEventArgs (string propertyName);
  public virtual string PropertyName { get; }
}
```

Observa que PropertyChangedEventArgs requiere el nombre de la propiedad que ha cambiado. Sin embargo, al aplicar el atributo [CallerMemberName], podemos implementar esta interfaz e invocar el evento sin especificar nunca los nombres de las propiedades:

```
public class Foo : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged = delegate { };

  void RaisePropertyChanged ([CallerMemberName] string propertyName = null)
    => PropertyChanged (this, new PropertyChangedEventArgs (propertyName));

  string customerName;
  public string CustomerName
  {
    get => customerName;
    set
    {
      if (value == customerName) return;
      customerName = value;
      RaisePropertyChanged();
      // The compiler converts the above line to:
      // RaisePropertyChanged ("CustomerName");
    }
  }
}
```

CallerArgumentExpression (C# 10)

Un parámetro de método al cual se aplica el atributo [CallerArgumentExpression] captura una expresión de argumento desde el lugar de la llamada:

```
Print (Math.PI * 2);

void Print (double number,
           [CallerArgumentExpression("number")] string expr = null)
  => Console.WriteLine (expr);

// Output: Math.PI * 2
```

El compilador proporciona literalmente el código fuente de la expresión de llamada, incluyendo comentarios:

```
Print (Math.PI /*(n)*/ * 2);

// Output:  Math.PI /*(n)*/ * 2
```

La aplicación principal de esta característica es al escribir bibliotecas de validación y aserción. En el siguiente ejemplo, se lanza una excepción cuyo mensaje incluye el texto "2 + 2 == 5". Esto ayuda en la depuración:

```
Assert (2 + 2 == 5);

void Assert (bool condition,
             [CallerArgumentExpression ("condition")] string message = null)
{
  if (!condition) throw new Exception ("Assertion failed: " + message);
}
```

Otro ejemplo es el método estático ThrowIfNull en la clase ArgumentNullException. Este método se introdujo en .NET 6 y se define de la siguiente manera:

```
public static void ThrowIfNull (object argument,
  [CallerArgumentExpression("argument")] string paramName = null)
{
  if (argument == null)
    throw new ArgumentNullException (paramName);
}
```

Se utiliza de la siguiente manera:

```
void Print (string message)
{
  ArgumentNullException.ThrowIfNull (message);
  ...
}
```

Puedes usar [CallerArgumentExpression] varias veces para capturar múltiples expresiones de argumentos.

Enlace dinámico

El enlace dinámico pospone el enlace, el proceso de resolver tipos, miembros y operadores, desde el tiempo de compilación hasta el tiempo de ejecución. El enlace dinámico es útil cuando en tiempo de compilación sabes que una cierta función, miembro u operación existe, pero el compilador no lo sabe. Esto ocurre comúnmente cuando trabajas con lenguajes dinámicos (como IronPython) y COM, así como en escenarios en los que de otro modo usarías reflexión.

Un tipo dinámico se declara con la palabra clave contextual dynamic:

```
dynamic d = GetSomeObject();
d.Quack();
```

Un tipo dinámico le indica al compilador que sea más flexible. Esperamos que el tipo en tiempo de ejecución de d tenga un método llamado Quack. Simplemente no podemos demostrarlo estáticamente. Debido a que d es dinámico, el compilador pospone el enlace de Quack a d hasta el tiempo de ejecución. Para entender lo que esto significa, es necesario distinguir entre enlace estático y enlace dinámico.

Enlace estático versus enlace dinámico

El ejemplo de enlace canónico es asignar un nombre a una función específica al compilar una expresión. Para compilar la siguiente expresión, el compilador necesita encontrar la implementación del método llamado Quack:

```
d.Quack();
```

Supongamos que el tipo estático de d es Duck:

```
Duck d = ...
d.Quack();
```

En el caso más simple, el compilador hace el enlace buscando un método sin parámetros llamado Quack en Duck. Si no lo encuentra, el compilador extiende la búsqueda a métodos con parámetros opcionales, métodos en las clases base de Duck y métodos de extensión que toman Duck como su primer parámetro. Si no se encuentra ninguna coincidencia, se producirá un error de compilación. Independientemente del método que se enlace, la conclusión es que el enlace lo realiza el compilador y el enlace depende por completo de conocer estáticamente los tipos de los operandos (en este caso, d). Esto lo hace un enlace estático.

Ahora cambiemos el tipo estático de d a object:

```
object d = ...
d.Quack();
```

Llamar a Quack nos dará un error de compilación, porque aunque el valor almacenado en d puede contener un método llamado Quack, el compilador no puede saberlo, porque la única información que tiene es el tipo de la variable, que en este caso es object. Pero cambiemos ahora el tipo estático de d a dynamic:

```
dynamic d = ...
d.Quack();
```

Un tipo dinámico es similar a object: ambos no proporcionan información descriptiva sobre un tipo. La diferencia es que te permite usarlo de formas que no se conocen en tiempo de compilación. Un objeto dinámico se enlaza en tiempo de ejecución en función de su tipo en tiempo de ejecución, no de su tipo en tiempo de compilación. Cuando el compilador ve una expresión vinculada de forma dinámica (que en general es una expresión que contiene cualquier valor de tipo dynamic), simplemente empaqueta la expresión para que el enlace se pueda hacer más tarde en tiempo de ejecución.

En tiempo de ejecución, si un objeto dinámico implementa IDynamicMetaObjectProvider, se utiliza esa interfaz para realizar el enlace. Si no lo hace, el enlace se produce de manera casi igual a como lo haría si el compilador conociera el tipo en tiempo de ejecución del objeto dinámico. Estas dos alternativas se denominan enlace personalizado y enlace de lenguaje.

Enlace personalizado

El enlace personalizado ocurre cuando un objeto dinámico implementa IDynamicMetaObjectProvider (IDMOP). Aunque puedes implementar IDMOP en tipos que escribas en C#, lo cual es útil hacer, el caso más común es que hayas adquirido un objeto IDMOP de un lenguaje dinámico implementado en .NET en el tiempo de ejecución de lenguajes dinámicos (DLR), como IronPython o IronRuby. Los objetos de esos lenguajes implementan implícitamente IDMOP como un medio para controlar directamente el significado de las operaciones realizadas en ellos.

Discutimos los vinculadores personalizados con más detalle en el Capítulo 19, pero por ahora, escribamos uno simple para demostrar la característica:

```
using System;
using System.Dynamic;

dynamic d = new Duck();
d.Quack();                  // Quack method was called
d.Waddle();                 // Waddle method was called

public class Duck : DynamicObject
{
  public override bool TryInvokeMember (
    InvokeMemberBinder binder, object[] args, out object result)
  {
    Console.WriteLine (binder.Name + " method was called");
    result = null;
    return true;
  }
}
```

La clase Duck en realidad no tiene un método Quack. En su lugar, utiliza el enlace personalizado para interceptar e interpretar todas las llamadas a métodos.

Enlace de lenguaje

El enlace de lenguaje ocurre cuando un objeto dinámico no implementa IDynamicMetaObjectProvider. Es útil cuando se trabaja con tipos imperfectamente diseñados o limitaciones inherentes en el sistema de tipos de .NET. Un problema típico al usar tipos numéricos es que no tienen una interfaz común. Hemos visto que podemos enlazar métodos de forma dinámica; lo mismo ocurre con los operadores:

```
int x = 3, y = 4;
Console.WriteLine (Mean (x, y));

dynamic Mean (dynamic x, dynamic y) => (x + y) / 2;
```

El beneficio es evidente: no necesitas duplicar el código para cada tipo numérico. Sin embargo, pierdes la seguridad de tipos estática, arriesgándote a excepciones en tiempo de ejecución en lugar de errores en tiempo de compilación.



El enlace dinámico elude la seguridad de tipos estática, pero no la seguridad de tipos en tiempo de ejecución. A diferencia de la reflexión (Capítulo 18), no puedes eludir las reglas de accesibilidad de miembros con el enlace dinámico.

Por diseño, el enlace de tiempo de ejecución de lenguajes se comporta de manera similar al enlace estático, como si los tipos de tiempo de ejecución de los objetos dinámicos se conocieran en tiempo de compilación. En nuestro ejemplo anterior, el comportamiento de nuestro programa sería idéntico si codificáramos Mean para que funcionara con el tipo int. La excepción más notable en la paridad entre el enlace estático y dinámico se encuentra en los métodos de extensión, que se discuten en "Funciones no llamables" en la página 238.



El enlace dinámico también incurre en una penalización de rendimiento. Sin embargo, debido a los mecanismos de almacenamiento en caché del DLR, las llamadas repetidas a la misma expresión dinámica se optimizan, lo que te permite llamar eficientemente a expresiones dinámicas en un bucle. Esta optimización reduce la sobrecarga típica para una expresión dinámica simple en el hardware actual a menos de 100 ns.

RuntimeBinderException

Si no se puede realizar el enlace de un miembro, se lanza una RuntimeBinderException. Puedes pensar en esto como un error en tiempo de compilación que ocurre en tiempo de ejecución:

```
dynamic d = 5;
d.Hello();                  // throws RuntimeBinderException
```

La excepción se lanza porque el tipo int no tiene un método Hello.

Representación en tiempo de ejecución de Dynamic

Existe una equivalencia profunda entre los tipos dynamic y object. El tiempo de ejecución trata la siguiente expresión como verdadera:

```
typeof (dynamic) == typeof (object)
```

Este principio se extiende a los tipos construidos y los tipos de matrices:

```
typeof (List<dynamic>) == typeof (List<object>)
typeof (dynamic[]) == typeof (object[])
```

Como una referencia a un objeto, una referencia dinámica puede apuntar a un objeto de cualquier tipo (excepto tipos de puntero):

```
dynamic x = "hello";
Console.WriteLine (x.GetType().Name);  // String

x = 123;  // No error (despite same variable)
Console.WriteLine (x.GetType().Name);  // Int32
```

Estructuralmente, no hay diferencia entre una referencia a un objeto y una referencia dinámica. Una referencia dinámica simplemente permite operaciones dinámicas en el objeto al que apunta. Puedes convertir de object a dynamic para realizar cualquier operación dinámica que desees en un objeto:

```
object o = new System.Text.StringBuilder();
dynamic d = o;
d.Append ("hello");
Console.WriteLine (o);   // hello
```



Si se refleja sobre un tipo que expone miembros dinámicos (públicos), se revela que esos miembros se representan como objetos anotados. Por ejemplo

```
public class Test
{
  public dynamic Foo;
}
```

Es equivalente a:

```
public class Test
{
  [System.Runtime.CompilerServices.DynamicAttribute]
  public object Foo;
}
```

Esto permite que los consumidores de ese tipo sepan que Foo debe tratarse como dinámico, al tiempo que permite que los lenguajes que no admiten el enlace dinámico utilicen object como alternativa.

Conversiones dinámicas

El tipo dynamic tiene conversiones implícitas desde y hacia todos los demás tipos:

```
int i = 7;
dynamic d = i;
long j = d;        // No cast required (implicit conversion)
```

Para que la conversión tenga éxito, el tipo de tiempo de ejecución del objeto dinámico debe ser implícitamente convertible al tipo estático de destino. El ejemplo anterior funcionó porque un int se puede convertir implícitamente en un long.

El siguiente ejemplo lanza una RuntimeBinderException porque un int no se puede convertir implícitamente en un short:

```
int i = 7;
dynamic d = i;
short j = d;      // throws RuntimeBinderException
```

var Versus dynamic

El tipo var y dynamic tienen una similitud superficial, pero la diferencia es profunda:

- var dice: "Deja que el compilador determine el tipo".
- dynamic dice: "Deja que el tiempo de ejecución determine el tipo".

Para ilustrar:

```
dynamic x = "hello";  // Static type is dynamic; runtime type is string
var y = "hello";      // Static type is string; runtime type is string
int i = x;            // Runtime error      (cannot convert string to int)
int j = y;            // Compile-time error (cannot convert string to int)
```

El tipo estático de una variable declarada con var puede ser dynamic:

```
dynamic x = "hello";
var y = x;            // Static type of y is dynamic
int z = y;            // Runtime error (cannot convert string to int)
```

Expresiones Dinámicas

Se pueden llamar dinámicamente campos, propiedades, métodos, eventos, constructores, indexadores, operadores y conversiones.

Intentar consumir el resultado de una expresión dinámica con un tipo de retorno void está prohibido, al igual que con una expresión de tipo estático. La diferencia es que el error ocurre en tiempo de ejecución:

```
dynamic list = new List<int>();
var result = list.Add (5);         // RuntimeBinderException thrown
```

Las expresiones que involucran operandos dinámicos suelen ser ellas mismas dinámicas debido al efecto de la falta de información de tipo, que se propaga en cascada:

```
dynamic x = 2;
var y = x * 3;       // Static type of y is dynamic
```

Hay un par de excepciones obvias a esta regla. Primero, la conversión de una expresión dinámica a un tipo estático produce una expresión estática:

```
dynamic x = 2;
var y = (int)x;      // Static type of y is int
```

Segundo, las invocaciones de constructores siempre producen expresiones estáticas, incluso cuando se llaman con argumentos dinámicos. En este ejemplo, x tiene un tipo estático de StringBuilder:

```
dynamic capacity = 10;
var x = new System.Text.StringBuilder (capacity);
```

Además, hay algunos casos particulares en los que una expresión que contiene un argumento dinámico es estática, incluyendo pasar un índice a un arreglo y expresiones de creación de delegados.

Llamadas dinámicas sin receptores dinámicos

El caso de uso canónico para dynamic involucra un receptor dinámico. Esto significa que un objeto dinámico es el receptor de una llamada de función dinámica:

```
dynamic x = ...;
x.Foo();          // x is the receiver
```

Sin embargo, también puedes llamar a funciones conocidas estáticamente con argumentos dinámicos. Estas llamadas están sujetas a la resolución de sobrecarga dinámica y pueden incluir lo siguiente:

- Métodos estáticos
- Constructores de instancia
- Métodos de instancia en receptores con un tipo conocido estáticamente

En el siguiente ejemplo, el Foo particular que se enlaza dinámicamente depende del tipo en tiempo de ejecución del argumento dinámico:

```
class Program
{
  static void Foo (int x)    => Console.WriteLine ("int");
  static void Foo (string x) => Console.WriteLine ("string");

  static void Main()
  {
    dynamic x = 5;
    dynamic y = "watermelon";

    Foo (x);    // int
    Foo (y);    // string
  }
}
```

Como no se involucra un receptor dinámico, el compilador puede realizar estáticamente una comprobación básica para ver si la llamada dinámica tendrá éxito. Verifica si existe una función con el nombre y número de parámetros correctos. Si no se encuentra ningún candidato, se produce un error en tiempo de compilación:

```
class Program
{
  static void Foo (int x)    => Console.WriteLine ("int");
  static void Foo (string x) => Console.WriteLine ("string");

  static void Main()
  {
    dynamic x = 5;
    Foo (x, x);        // Compiler error - wrong number of parameters
    Fook (x);          // Compiler error - no such method name
  }
}
```

Tipos estáticos en expresiones dinámicas

Es obvio que se utilizan tipos dinámicos en la vinculación dinámica. No es tan obvio que también se utilizan tipos estáticos siempre que sea posible en la vinculación dinámica. Considera lo siguiente:

```
class Program
{
  static void Foo (object x, object y) { Console.WriteLine ("oo"); }
  static void Foo (object x, string y) { Console.WriteLine ("os"); }
  static void Foo (string x, object y) { Console.WriteLine ("so"); }
  static void Foo (string x, string y) { Console.WriteLine ("ss"); }

  static void Main()
  {
    object o = "hello";
    dynamic d = "goodbye";
    Foo (o, d);               // os
  }
}
```

La llamada a Foo (o,d) se enlaza dinámicamente porque uno de sus argumentos, d, es dinámico. Pero debido a que o se conoce estáticamente, la vinculación, aunque ocurre dinámicamente, hará uso de eso. En este caso, la resolución de sobrecarga seleccionará la segunda implementación de Foo debido al tipo estático de o y el tipo en tiempo de ejecución de d.

En otras palabras, el compilador es "tan estático como sea posible".

Funciones no llamables

Algunas funciones no pueden ser llamadas de forma dinámica. No puedes llamar a lo siguiente:

- Métodos de extensión (mediante la sintaxis de método de extensión)
- Miembros de una interfaz, si es necesario realizar una conversión a esa interfaz para hacerlo
- Miembros base ocultados por una subclase

Comprender por qué es útil para entender la vinculación dinámica.

La vinculación dinámica requiere dos piezas de información: el nombre de la función a llamar y el objeto sobre el cual llamar la función. Sin embargo, en cada uno de los tres escenarios no llamables, hay un tipo adicional involucrado, que solo se conoce en tiempo de compilación. Hasta la fecha de esta escritura, no hay forma de especificar estos tipos adicionales de forma dinámica.

Al llamar a métodos de extensión, ese tipo adicional es implícito. Es la clase estática en la que se define el método de extensión. El compilador la busca según las directivas using en tu código fuente. Esto hace que los métodos de extensión sean conceptos exclusivos de tiempo de compilación porque las directivas using desaparecen al compilarse (después de haber realizado su trabajo en el proceso de vinculación al mapear nombres simples a nombres calificados por espacio de nombres).

Al llamar a miembros a través de una interfaz, especificas ese tipo adicional mediante una conversión implícita o explícita. Hay dos escenarios en los que es posible que desees hacer esto: al llamar a miembros de una interfaz implementados explícitamente y al llamar a miembros de una interfaz implementados en un tipo interno de otra assembly. Podemos ilustrar lo primero con los siguientes dos tipos:

```
interface IFoo   { void Test();        }
class Foo : IFoo { void IFoo.Test() {} }
```

Para llamar al método Test, debemos hacer una conversión al tipo IFoo. Esto es fácil con tipado estático:

```
IFoo f = new Foo();   // Implicit cast to interface
f.Test();
```

Ahora considera la situación con tipado dinámico:

```
IFoo f = new Foo();
dynamic d = f;
d.Test();             // Exception thrown
```

La conversión implícita mostrada en negrita le indica al compilador que vincule las llamadas subsiguientes a miembros en f a IFoo en lugar de Foo, es decir, que vea ese objeto a través de la lente de la interfaz IFoo. Sin embargo, esa lente se pierde en tiempo de ejecución, por lo que el DLR no puede completar la vinculación. La pérdida se ilustra de la siguiente manera:

```
Console.WriteLine (f.GetType().Name);    // Foo
```

Una situación similar surge al llamar a un miembro base oculto: debes especificar un tipo adicional mediante una conversión o la palabra clave base, y ese tipo adicional se pierde en tiempo de ejecución.



Si necesitas invocar miembros de una interfaz de forma dinámica, una solución alternativa es utilizar la biblioteca de código abierto Uncapsulator, disponible en NuGet y GitHub. Uncapsulator fue escrita por el autor para abordar este problema y utiliza la vinculación personalizada para proporcionar una alternativa mejor que el uso de dynamic:

```
IFoo f = new Foo();
dynamic uf = f.Uncapsulate();
uf.Test();
```

Uncapsulator también te permite hacer conversiones a tipos base e interfaces por nombre, llamar dinámicamente a miembros estáticos y acceder a miembros no públicos de un tipo.

Sobrecarga de operadores

Puedes sobrecargar operadores para proporcionar una sintaxis más natural para tipos personalizados. La sobrecarga de operadores se utiliza más apropiadamente para implementar estructuras personalizadas que representan tipos de datos bastante primitivos. Por ejemplo, un tipo numérico personalizado es un candidato excelente para la sobrecarga de operadores.

Los siguientes operadores simbólicos pueden ser sobrecargados:

+ (unary)	- (unary)	!	~	++
--	+	-	*	/
%	&	|	^	<<
>>	==	!=	>	<
>=	<=			

Los siguientes operadores también pueden ser sobrecargados:

- Conversiones implícitas y explícitas (con las palabras clave implicit y explicit).
- Los operadores true y false (no literales).

Los siguientes operadores son sobrecargados indirectamente:

- Los operadores de asignación compuesta (por ejemplo, +=, /=) son implícitamente reemplazados al reemplazar los operadores no compuestos (por ejemplo, +, /).
- Los operadores condicionales && y || son implícitamente reemplazados al reemplazar los operadores bit a bit & y |.

Funciones de operador

Sobrecargas un operador declarando una función de operador. Una función de operador tiene las siguientes reglas:

- El nombre de la función se especifica con la palabra clave operator seguida del símbolo del operador.
- La función de operador debe ser estática y pública.
- Los parámetros de la función de operador representan los operandos.
- El tipo de retorno de una función de operador representa el resultado de una expresión.
- Al menos uno de los operandos debe ser del tipo en el que se declara la función de operador.

En el siguiente ejemplo, definimos una estructura llamada Note que representa una nota musical y luego sobrecargamos el operador +:

```
public struct Note
{
  int value;
  public Note (int semitonesFromA) { value = semitonesFromA; }
  public static Note operator + (Note x, int semitones)
  {
    return new Note (x.value + semitones);

   }
 }
```

Esta sobrecarga nos permite sumar un entero a una nota:

```
Note B = new Note (2);
Note CSharp = B + 2;
```

La sobrecarga de un operador automáticamente sobrecarga el operador de asignación compuesta correspondiente. En nuestro ejemplo, como hemos sobrecargado +, también podemos usar +=:

```
CSharp += 2;
```

Al igual que con los métodos y propiedades, C# permite escribir funciones de operador que consisten en una sola expresión de forma más concisa con la sintaxis de cuerpo de expresión:

```
public static Note operator + (Note x, int semitones)
                              => new Note (x.value + semitones);
```

Sobrecarga de operadores de igualdad y comparación

A veces se sobrecargan los operadores de igualdad y comparación al escribir estructuras y en casos raros al escribir clases. Estos operadores tienen reglas y obligaciones especiales que explicaremos en el Capítulo 6. Un resumen de estas reglas es el siguiente:

Parejas

El compilador de C# exige que los operadores que son pares lógicos, como (== !=), (< >) y (<= >=), estén definidos juntos.

Equals y GetHashCode

En la mayoría de los casos, si sobrecargas (==) y (!=), debes anular los métodos Equals y GetHashCode definidos en object para obtener un comportamiento significativo. El compilador de C# mostrará una advertencia si no lo haces. (Consulta "Comparación de igualdad" en la página 324 para obtener más detalles).

IComparable e IComparable<T>

Si sobrecargas (< >) y (<= >=), debes implementar IComparable e IComparable<T>.

Conversiones implícitas y explícitas personalizadas

Las conversiones implícitas y explícitas son operadores sobrecargables. Estas conversiones se sobrecargan típicamente para facilitar la conversión entre tipos fuertemente relacionados (como tipos numéricos) de manera concisa y natural.

Para convertir entre tipos débilmente relacionados, se recomiendan las siguientes estrategias:

- Escriba un constructor que tenga un parámetro del tipo desde el cual se realizará la conversión.
- Escriba métodos ToXXX y FromXXX (estáticos) para convertir entre tipos.

Como se explica en la discusión sobre tipos, la razón detrás de las conversiones implícitas es que se garantiza que tengan éxito y no pierdan información durante la conversión. Por otro lado, una conversión explícita debe ser requerida cuando las circunstancias en tiempo de ejecución determinen si la conversión tendrá éxito o si se puede perder información durante la conversión.

En este ejemplo, definimos conversiones entre nuestro tipo musical "Note" y un tipo "double" (que representa la frecuencia en hertz de esa nota):

```
...
// Convert to hertz
public static implicit operator double (Note x)
  => 440 * Math.Pow (2, (double) x.value / 12 );

// Convert from hertz (accurate to the nearest semitone)
public static explicit operator Note (double x)
  => new Note ((int) (0.5 + 12 * (Math.Log (x/440) / Math.Log(2) ) ));
...

Note n = (Note)554.37;  // explicit conversion
double x = n;           // implicit conversion
```



Siguiendo nuestras propias pautas, este ejemplo podría implementarse mejor con un método ToFrequency (y un método estático FromFrequency) en lugar de utilizar operadores implícitos y explícitos.



Las conversiones personalizadas son ignoradas por los operadores as e is:

```
Console.WriteLine (554.37 is Note);   // False
Note n = 554.37 as Note;              // Error
```

Sobrecargar los operadores true y false

Los operadores true y false se sobrecargan en casos extremadamente raros de tipos que son "Booleanos en espíritu" pero que no tienen una conversión a bool. Un ejemplo es un tipo que implementa la lógica de tres estados: al sobrecargar true y false, dicho tipo puede funcionar sin

problemas con declaraciones y operadores condicionales, como if, do, while, for, &&, || y ?: La estructura System.Data.SqlTypes.SqlBoolean proporciona esta funcionalidad:

```
SqlBoolean a = SqlBoolean.Null;
if (a)
  Console.WriteLine ("True");
else if (!a)
  Console.WriteLine ("False");
else
  Console.WriteLine ("Null");

OUTPUT:
Null
```

El siguiente código es una reimplementación de las partes de SqlBoolean necesarias para demostrar los operadores true y false:

```
public struct SqlBoolean
{
  public static bool operator true (SqlBoolean x)
    => x.m_value == True.m_value;

  public static bool operator false (SqlBoolean x)
    => x.m_value == False.m_value;

  public static SqlBoolean operator ! (SqlBoolean x)
  {
    if (x.m_value == Null.m_value)  return Null;
    if (x.m_value == False.m_value) return True;
    return False;
  }

  public static readonly SqlBoolean Null =  new SqlBoolean(0);
  public static readonly SqlBoolean False = new SqlBoolean(1);
  public static readonly SqlBoolean True =  new SqlBoolean(2);

  private SqlBoolean (byte value) { m_value = value; }
  private byte m_value;
}
```

Código no seguro y punteros

C# admite la manipulación directa de memoria a través de punteros dentro de bloques de código marcados como no seguros y compilados con la opción de compilador /unsafe. Los tipos de puntero son principalmente útiles para la interoperabilidad con API de C, pero también se pueden usar para acceder a memoria fuera del montón administrado o para puntos críticos de rendimiento.

Conceptos básicos de punteros

Para cada tipo de valor o tipo de referencia V, hay un tipo de puntero correspondiente V*. Una instancia de puntero contiene la dirección de una variable. Los tipos de puntero pueden ser

convertidos (de forma insegura) a cualquier otro tipo de puntero. A continuación, se presentan los principales operadores de puntero:

Operator	Meaning
&	The *address-of* operator returns a pointer to the address of a variable.
*	The *dereference* operator returns the variable at the address of a pointer.
->	The *pointer-to-member* operator is a syntactic shortcut, in which `x->y` is equivalent to `(*x).y`.

De acuerdo con C, al agregar (o restar) un desplazamiento entero a un puntero se genera otro puntero. Al restar un puntero de otro se genera un entero de 64 bits (tanto en plataformas de 64 bits como de 32 bits).

Código no seguro

Al marcar un tipo, miembro de tipo o bloque de instrucciones con la palabra clave unsafe, se te permite utilizar tipos de puntero y realizar operaciones de puntero al estilo de C en la memoria dentro de ese ámbito. Aquí tienes un ejemplo de cómo utilizar punteros para procesar rápidamente un mapa de bits:

```
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
      *p++ &= 0xFF;
  }
}
```

El código no seguro puede ejecutarse más rápido que una implementación segura correspondiente. En este caso, el código habría requerido un bucle anidado con indexación de matrices y verificación de límites. Un método C# no seguro también puede ser más rápido que llamar a una función externa de C siempre que no haya sobrecarga asociada con salir del entorno de ejecución administrado.

La instrucción `fixed`

La instrucción `fixed` se requiere para fijar un objeto administrado, como el mapa de bits en el ejemplo anterior. Durante la ejecución de un programa, se asignan y desasignan muchos objetos del montón. Para evitar un desperdicio o fragmentación innecesarios de memoria, el recolector de basura mueve los objetos. Apuntar a un objeto es inútil si su dirección podría cambiar mientras se referencia, por lo que la instrucción `fixed` le indica al recolector de basura que "fije" el objeto y no lo mueva. Esto puede tener un impacto en la eficiencia del tiempo de ejecución, por lo que debes usar bloques `fixed` solo brevemente y evitar la asignación en el montón dentro del bloque `fixed`.

Dentro de una instrucción `fixed`, puedes obtener un puntero a cualquier tipo de valor, a una matriz de tipos de valor o a una cadena. En el caso de las matrices y las cadenas, el puntero apuntará realmente al primer elemento, que es un tipo de valor.

Los tipos de valor declarados en línea dentro de tipos de referencia requieren que el tipo de referencia se fije, de la siguiente manera:

```
Test test = new Test();
unsafe
{
  fixed (int* p = &test.X)   // Pins test
  {
    *p = 9;
  }
  Console.WriteLine (test.X);
}

class Test { public int X; }
```

Describimos la instrucción `fixed` con más detalle en "Asignación de una estructura a memoria no administrada" en la página 973.

El operador Pointer-to-Member

Además de los operadores & y *, C# también proporciona el operador -> al estilo de C++, que se puede utilizar en structs:

```
Test test = new Test();
unsafe
{
  Test* p = &test;
  p->X = 9;
  System.Console.WriteLine (test.X);
}

struct Test { public int X; }
```

La palabra clave stackalloc

Puedes asignar memoria en un bloque en la pila de forma explícita utilizando la palabra clave stackalloc. Debido a que se asigna en la pila, su tiempo de vida se limita a la ejecución del método, al igual que con cualquier otra variable local (cuya vida no se ha extendido por ser capturada por una expresión lambda, bloque de iterador o función asíncrona). El bloque puede utilizar el operador [] para indexar la memoria:

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
   Console.WriteLine (a[i]);
```

En el Capítulo 23, describimos cómo puedes utilizar Span<T> para gestionar memoria asignada en la pila sin utilizar la palabra clave unsafe:

```
Span<int> a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
  Console.WriteLine (a[i]);
```

Buffers de tamaño fijo

La palabra clave fixed tiene otro uso, que es crear buffers de tamaño fijo dentro de structs (esto puede ser útil al llamar a una función no administrada, consulta el Capítulo 24):

```
new UnsafeClass ("Christian Troy");

unsafe struct UnsafeUnicodeString
{
  public short Length;
  public fixed byte Buffer[30];   // Allocate block of 30 bytes
}

unsafe class UnsafeClass
{
  UnsafeUnicodeString uus;

  public UnsafeClass (string s)
  {
    uus.Length = (short)s.Length;

    fixed (byte* p = uus.Buffer)
      for (int i = 0; i < s.Length; i++)
        p[i] = (byte) s[i];
  }
}
```

Buffers de tamaño fijo no son matrices: si Buffer fuera una matriz, consistiría en una referencia a un objeto almacenado en el montón (heap) administrado, en lugar de 30 bytes dentro de la estructura en sí.

La palabra clave fixed también se utiliza en este ejemplo para fijar en su lugar el objeto en el montón (heap) que contiene el buffer (que será la instancia de la clase UnsafeClass). Por lo tanto, fixed tiene dos significados diferentes: fijo en tamaño y fijo en lugar. Ambos se utilizan a menudo juntos, ya que un buffer de tamaño fijo debe estar fijo en su lugar para poder ser utilizado.

void*

Un puntero void (void*) no hace suposiciones sobre el tipo de los datos subyacentes y es útil para funciones que trabajan con memoria sin procesar. Existe una conversión implícita de cualquier tipo de puntero a void*. Un puntero void* no se puede desreferenciar y no se pueden realizar operaciones aritméticas en punteros void. Aquí tienes un ejemplo:

```
short[] a = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
unsafe
{
  fixed (short* p = a)
  {
    //sizeof returns size of value-type in bytes
    Zap (p, a.Length * sizeof (short));
  }
}
foreach (short x in a)
  System.Console.WriteLine (x);   // Prints all zeros

unsafe void Zap (void* memory, int byteCount)
{
  byte* b = (byte*)memory;
  for (int i = 0; i < byteCount; i++)
    *b++ = 0;
}
```

Enteros de tamaño nativo

Los tipos de enteros nint y nuint de tamaño nativo (introducidos en C# 9) tienen un tamaño que coincide con el espacio de direcciones del proceso en tiempo de ejecución (en la práctica, 32 o 64 bits). Los enteros de tamaño nativo pueden mejorar la eficiencia, la seguridad contra desbordamiento y la comodidad al realizar operaciones aritméticas con punteros.

La mejora en la eficiencia surge porque cuando restas dos punteros en C#, el resultado siempre es un entero de 64 bits (long), lo cual es ineficiente en plataformas de 32 bits. Al convertir primero los punteros a nint, el resultado de la resta también será de tipo nint (que será de 32 bits en una plataforma de 32 bits):

```
unsafe nint AddressDif (char* x, char* y) => (nint)x - (nint)y;
```

La ganancia en seguridad contra desbordamiento y comodidad surge cuando necesitas un tipo para representar un desplazamiento en memoria o la longitud de un buffer. Esto se debe a que la alternativa histórica al uso de nint/nuint ha sido reutilizar los tipos System.IntPtr y System.UIntPtr, cuyo propósito principal es envolver identificadores del sistema operativo o punteros de dirección, permitiendo la interoperabilidad fuera de un contexto inseguro. Aunque tienen el tamaño nativo, estos tipos tienen un soporte limitado para operaciones aritméticas, y el soporte que tienen siempre es sin comprobación de desbordamiento (por lo que los desbordamientos fallan silenciosamente).

En contraste, los enteros de tamaño nativo se comportan de manera similar a los enteros estándar, con soporte completo para operaciones aritméticas y comprobación de desbordamiento:

```
nint x = 123, y = 234;
checked
{
  nint sum = x + y, product = x * y;
  Console.WriteLine (product);
}
```

Los enteros de tamaño nativo pueden recibir constantes de enteros de 32 bits (pero no constantes de enteros de 64 bits, ya que estos podrían desbordarse en tiempo de ejecución). Puedes utilizar una conversión explícita para convertir entre ellos y otros tipos enteros.

En tiempo de ejecución, nint y nuint se mapean a las estructuras IntPtr y UIntPtr, por lo que puedes convertir entre ellos sin necesidad de una conversión explícita (una conversión de identidad):

```
nint x = 123;
IntPtr p = x;
nint y = p;
```

Por las razones descritas anteriormente, nint/nuint no son simplemente atajos para IntPtr/UIntPtr, a pesar de su equivalencia en tiempo de ejecución. Específicamente, el compilador trata una variable de tipo nint/nuint como un tipo numérico, lo que te permite realizar operaciones aritméticas que no están implementadas por IntPtr y UIntPtr (y con bloques checked respetados).



Una variable nint/nuint es como un IntPtr/UIntPtr con un sombrero especial. Este sombrero es reconocido por el compilador y significa "por favor, trátame como un tipo numérico seguro".

Este comportamiento de sombrero múltiple es único de los enteros de tamaño nativo. Por ejemplo, int actúa como un sinónimo puro de System.Int32, y ambos pueden intercambiarse libremente.

Esta no equivalencia significa que ambas construcciones son útiles:

- nint/nuint son útiles para representar un desplazamiento de memoria o la longitud de un buffer.
- IntPtr/UIntPtr son útiles para envolver identificadores y punteros para interoperabilidad.

El uso de los tipos de esta manera también señala correctamente tu intención.



Un buen ejemplo del uso real de nint y nuint es en la implementación de Buffer.MemoryCopy. Puedes ver esto en el código fuente de .NET para Buffer.cs en GitHub o al descompilar el método en ILSpy. Una versión simplificada también se ha incluido en los ejemplos de LINQPad para C# 10 en un libro llamado "C# 10 en pocas palabras".

Punteros a funciones

Los punteros a funciones (a partir de C# 9) son similares a los delegados, pero sin la indirección de una instancia de delegado; en cambio, apuntan directamente a un método. Un puntero a función puede apuntar solo a métodos estáticos, carece de capacidad de multicast y requiere un contexto inseguro (ya que omite la seguridad de tipos en tiempo de ejecución). Su principal propósito es simplificar y optimizar la interoperabilidad con APIs no administradas (consulta "Callbacks from Unmanaged Code" en la página 967).

Se declara un tipo de puntero a función de la siguiente manera (con el tipo de retorno apareciendo al final):

```
delegate*<int, char, string, void>   // (void refers to the return type)
```

Esto coincide con una función con esta firma:

void AlgunaFuncion(int x, char y, string z)

El operador & crea un puntero a función a partir de un grupo de métodos. Aquí tienes un ejemplo completo:

```
unsafe
{
  delegate*<string, int> functionPointer = &GetLength;
  int length = functionPointer ("Hello, world");

  static int GetLength (string s) => s.Length;
}
```

En este ejemplo, `punteroFuncion` no es un objeto en el que se pueda llamar a un método como Invoke (o con una referencia a un objeto Target). En cambio, es una variable que apunta directamente a la dirección de memoria del método objetivo:

```
Console.WriteLine ((IntPtr)functionPointer);
```

Al igual que cualquier otro puntero, no está sujeto a comprobación de tipos en tiempo de ejecución. Lo siguiente trata el valor de retorno de nuestra función como un decimal (que, al ser más largo que un int, significa que incorporamos algo de memoria aleatoria en la salida):

```
var pointer2 = (delegate*<string, decimal>) (IntPtr) functionPointer;
Console.WriteLine (pointer2 ("Hello, unsafe world"));
```

[SkipLocalsInit]

Cuando C# compila un método, emite una bandera que indica al tiempo de ejecución que inicialice las variables locales del método a sus valores predeterminados (mediante la anulación de la memoria).

A partir de C# 9, puedes indicar al compilador que no emita esta bandera aplicando el atributo [SkipLocalInit] a un método (en el espacio de nombres System.Runtime.CompilerServices):

```
[SkipLocalsInit]
void Foo() ...
```

También puedes aplicar este atributo a un tipo, lo cual es equivalente a aplicarlo a todos los métodos del tipo, o incluso a un módulo completo (el contenedor de un ensamblado):

```
[module: System.Runtime.CompilerServices.SkipLocalsInit]
```

En escenarios normales y seguros, [SkipLocalsInit] tiene poco efecto en la funcionalidad o rendimiento, debido a que la política de asignación definitiva de C# requiere que asignes explícitamente valores a las variables locales antes de poder leerlas. Esto significa que el optimizador JIT es probable que emita el mismo código máquina, ya sea que se aplique o no el atributo.

Sin embargo, en un contexto inseguro, el uso de [SkipLocalsInit] puede ahorrarle al CLR la sobrecarga de inicializar variables locales de tipo de valor, lo que genera una pequeña ganancia de rendimiento en métodos que hacen un uso extensivo de la pila (a través de un gran stackalloc). El siguiente ejemplo imprime memoria no inicializada cuando se aplica [SkipLocalsInit] (en lugar de imprimir ceros en todos los casos):

```
[SkipLocalsInit]
unsafe void Foo()
{
  int local;
  int* ptr = &local;
  Console.WriteLine (*ptr);

  int* a = stackalloc int [100];
  for (int i = 0; i < 100; ++i) Console.WriteLine (a [i]);
}
```

Curiosamente, puedes lograr el mismo resultado en un contexto "seguro" mediante el uso de Span<T>:

```
[SkipLocalsInit]
void Foo()
{
  Span<int> a = stackalloc int [100];
  for (int i = 0; i < 100; ++i) Console.WriteLine (a [i]);
}
```

En consecuencia, el uso de [SkipLocalsInit] requiere que compile tu ensamblado con la opción unsafe, incluso si ninguno de tus métodos está marcado como unsafe.

Directivas del preprocesador

Las directivas del preprocesador proporcionan al compilador información adicional sobre regiones de código. Las directivas del preprocesador más comunes son las directivas condicionales, que ofrecen una forma de incluir o excluir regiones de código de la compilación:

```
#define DEBUG
class MyClass
{
  int x;
  void Foo()
  {
    #if DEBUG
    Console.WriteLine ("Testing: x = {0}", x);
    #endif
  }
  ...
}
```

En esta clase, la declaración en Foo se compila condicionalmente en función de la presencia del símbolo DEBUG. Si eliminamos el símbolo DEBUG, la declaración no se compilará. Puedes definir símbolos del preprocesador dentro de un archivo fuente (como hemos hecho aquí) o a nivel de proyecto en el archivo .csproj:

```
<PropertyGroup>
  <DefineConstants>DEBUG;ANOTHERSYMBOL</DefineConstants>
</PropertyGroup>
```

Con las directivas #if y #elif, puedes utilizar los operadores ||, && y ! para realizar operaciones de or, and y not en múltiples símbolos. La siguiente directiva instruye al compilador que incluya el código que sigue si el símbolo TESTMODE está definido y el símbolo DEBUG no está definido:

```
#if TESTMODE && !DEBUG
  ...
```

Sin embargo, debes tener en cuenta que no estás construyendo una expresión C# ordinaria y los símbolos en los que operas no tienen ninguna conexión con variables, estáticas o de otro tipo.

Las directivas #error y #warning evitan el uso incorrecto accidental de las directivas condicionales al hacer que el compilador genere una advertencia o error en función de un conjunto indeseable de símbolos de compilación. La Tabla 4-1 muestra las directivas del preprocesador.

Table 4-1. Preprocessor directives

Preprocessor directive	Action
`#define` *`symbol`*	Defines *`symbol`*
`#undef` *`symbol`*	Undefines *`symbol`*
`#if` *`symbol`* [*`operator symbol2`*]...	*`symbol`* to test *`operator`*s are ==, !=, &&, and ||, followed by `#else`, `#elif`, and `#endif`
`#else`	Executes code to subsequent `#endif`
`#elif` *`symbol`* [*`operator symbol2`*]	Combines `#else` branch and `#if` test

Preprocessor directive	Action
`#endif`	Ends conditional directives
`#warning text`	*text* of the warning to appear in compiler output
`#error text`	*text* of the error to appear in compiler output
`#error version`	Reports the compiler version and exits
`#pragma warning [disable | restore]`	Disables/restores compiler warning(s)
`#line [ number ["file"] | hidden]`	*number* specifies the line in source code (a column can also be specified from C# 10); *file* is the filename to appear in computer output; `hidden` instructs debuggers to skip over code from this point until the next `#line` directive
`#region name`	Marks the beginning of an outline
`#endregion`	Ends an outline region
`#nullable option`	See "Nullable reference types" on page 16

Atributos condicionales

Un atributo decorado con el atributo Conditional se compilará solo si está presente un determinado símbolo del preprocesador:

```
// file1.cs
#define DEBUG
using System;
using System.Diagnostics;
[Conditional("DEBUG")]
public class TestAttribute : Attribute {}

// file2.cs
#define DEBUG
[Test]
class Foo
{
  [Test]
  string s;
}
```

El compilador incorporará los atributos [Test] solo si el símbolo DEBUG está en ámbito para file2.cs.

Advertencia Pragma

El compilador genera una advertencia cuando detecta algo en tu código que parece ser involuntario. A diferencia de los errores, las advertencias no impiden que tu aplicación se compile.

Las advertencias del compilador pueden ser extremadamente valiosas para detectar errores. Sin embargo, su utilidad se ve socavada cuando se obtienen advertencias falsas. En una aplicación grande, mantener una buena relación señal-ruido es esencial para poder notar las advertencias "reales". Con este fin, el compilador te permite suprimir selectivamente las advertencias

mediante el uso de la directiva #pragma warning. En este ejemplo, le indicamos al compilador que no nos advierta sobre el campo "Message" que no se está utilizando:

```
public class Foo
{
  static void Main() { }

  #pragma warning disable 414
  static string Message = "Hello";
  #pragma warning restore 414
}
```

Omitir el número en la directiva #pragma warning desactiva o restaura todos los códigos de advertencia.

Si eres meticuloso al aplicar esta directiva, puedes compilar con el interruptor /warnaserror, lo que indica al compilador que trate cualquier advertencia residual como un error.

Documentación XML

Un comentario de documentación es un fragmento de XML incrustado que documenta un tipo o miembro. Un comentario de documentación se coloca inmediatamente antes de una declaración de tipo o miembro y comienza con tres barras inclinadas:

```
/// <summary>Cancels a running query.</summary>
public void Cancel() { ... }
```

Los comentarios multilínea se pueden hacer de la siguiente manera:

```
/// <summary>
/// Cancels a running query
/// </summary>
public void Cancel() { ... }
```

O así (nota la estrella adicional al principio):

```
/**
    <summary> Cancels a running query. </summary>
*/
public void Cancel() { ... }
```

Si agregas la siguiente opción a tu archivo .csproj

```
<PropertyGroup>
  <DocumentationFile>SomeFile.xml</DocumentationFile>
</PropertyGroup>
```

el compilador extraerá y recopilará los comentarios de documentación en el archivo XML especificado. Esto tiene dos usos principales:

- Si se coloca en la misma carpeta que el ensamblado compilado, herramientas como Visual Studio y LINQPad leen automáticamente el archivo XML y utilizan la información para proporcionar listas de miembros IntelliSense a los consumidores del mismo ensamblado.

- Herramientas de terceros (como Sandcastle y NDoc) pueden transformar el archivo XML en un archivo de ayuda HTML.

Etiquetas de documentación XML estándar

Estas son las etiquetas XML estándar que Visual Studio y los generadores de documentación reconocen:

<summary>

```
<summary>...</summary>
```

Indica el mensaje que la herramienta IntelliSense debe mostrar para el tipo o miembro, típicamente una frase o una oración.

<remarks>

```
<remarks>...</remarks>
```

Texto adicional que describe el tipo o miembro. Los generadores de documentación recogen esto y lo incorporan a la descripción del tipo o miembro.

<param>

```
<param name="name">...</param>
```

Explica un parámetro de un método.

<returns>

```
<returns>...</returns>
```

Explica el valor de retorno de un método.

<exception>

```
<exception [cref="type"]>...</exception>
```

Enumera una excepción que un método puede lanzar (cref se refiere al tipo de excepción).

<example>

```
<example>...</example>
```

Indica un ejemplo (utilizado por los generadores de documentación). Esto generalmente contiene texto descriptivo y código fuente (el código fuente se encuentra típicamente dentro de una etiqueta <c> o <code>).

<c>

```
<c>...</c>
```

Indica un fragmento de código en línea. Esta etiqueta se usa generalmente dentro de un bloque <example>.

<code>

```
<code>...</code>
```

Indica un ejemplo de código en varias líneas. Esta etiqueta se usa generalmente dentro de un bloque <example>.

<see>

```
<see cref="member">...</see>
```

Inserta una referencia cruzada en línea a otro tipo o miembro. Los generadores de documentación HTML suelen convertir esto en un hipervínculo. El compilador emite una advertencia si el nombre del tipo o miembro es inválido. Para hacer referencia a tipos genéricos, utiliza llaves; por ejemplo, cref="Foo{T,U}".

<seealso>

```
<seealso cref="member">...</seealso>
```

Hace referencia a otro tipo o miembro. Los generadores de documentación suelen escribir esto en una sección separada de "Ver también" al final de la página.

<paramref>

```
<paramref name="name"/>
```

Hace referencia a un parámetro desde dentro de una etiqueta <summary> o <remarks>.

```
<list>
        <list type=[ bullet | number | table ]>
          <listheader>
            <term>...</term>
            <description>...</description>
          </listheader>
          <item>
            <term>...</term>
            <description>...</description>
          </item>
        </list>
```

Instruye a los generadores de documentación a emitir una lista con viñetas, numerada o en formato de tabla.

<para>

```
<para>...</para>
```

Instruye a los generadores de documentación a dar formato al contenido en un párrafo separado.

<include>

```
<include file='filename' path='tagpath[@name="id"]'>...</include>
```

Combina un archivo XML externo que contiene documentación. El atributo path indica una consulta XPath a un elemento específico en ese archivo.

Etiquetas definidas por el usuario

No hay nada especial en las etiquetas XML predefinidas reconocidas por el compilador de C#, y eres libre de definir las tuyas propias. El único procesamiento especial realizado por el compilador es en la etiqueta <param> (en la que verifica el nombre del parámetro y que todos los parámetros del método estén documentados) y en el atributo cref (en el que verifica que el atributo se refiere a un tipo o miembro real y lo expande a un ID de tipo o miembro completamente calificado). También puedes usar el atributo cref en tus propias etiquetas; se verifica y se expande de la misma manera que en las etiquetas predefinidas <exception>, <permission>, <see> y <seealso>.

Referencias cruzadas de tipos o miembros

Los nombres de tipos y las referencias cruzadas de tipos o miembros se traducen en ID que define de manera única el tipo o miembro. Estos nombres se componen de un prefijo que define lo que representa el ID y una firma del tipo o miembro. A continuación se presentan los prefijos de miembros:

XML type prefix	ID prefixes applied to…
N	Namespace
T	Type (class, struct, enum, interface, delegate)
F	Field
P	Property (includes indexers)
M	Method (includes special methods)
E	Event
!	Error

Las reglas que describen cómo se generan las firmas están bien documentadas, aunque son bastante complejas.

Aquí tienes un ejemplo de un tipo y los IDs que se generan:

```
// Namespaces do not have independent signatures
namespace NS
{
  /// T:NS.MyClass
  class MyClass
  {
    /// F:NS.MyClass.aField
    string aField;

    /// P:NS.MyClass.aProperty
    short aProperty {get {...} set {...}}

    /// T:NS.MyClass.NestedType
    class NestedType {...};

    /// M:NS.MyClass.X()
```

```
    void X() {...}

    /// M:NS.MyClass.Y(System.Int32,System.Double@,System.Decimal@)
    void Y(int p1, ref double p2, out decimal p3) {...}

    /// M:NS.MyClass.Z(System.Char[ ],System.Single[0:,0:])
    void Z(char[ ] p1, float[,] p2) {...}

    /// M:NS.MyClass.op_Addition(NS.MyClass,NS.MyClass)
    public static MyClass operator+(MyClass c1, MyClass c2) {...}

    /// M:NS.MyClass.op_Implicit(NS.MyClass)~System.Int32
    public static implicit operator int(MyClass c) {...}

    /// M:NS.MyClass.#ctor
    MyClass() {...}

    /// M:NS.MyClass.Finalize
    ~MyClass() {...}

    /// M:NS.MyClass.#cctor
    static MyClass() {...}
  }
}
```

5 Visión general de .NET

Casi todas las capacidades del tiempo de ejecución .NET 6 se exponen a través de un vasto conjunto de tipos administrados. Estos tipos se organizan en espacios de nombres jerárquicos y se empaquetan en un conjunto de ensamblados.

Algunos de los tipos de .NET se utilizan directamente por el CLR y son esenciales para el entorno de hospedaje administrado. Estos tipos residen en un ensamblado llamado System.Private.CoreLib.dll e incluyen los tipos incorporados de C#, así como las clases de colección básicas y los tipos para procesamiento de secuencias, serialización, reflexión, subprocesamiento y interoperabilidad nativa.



System.Private.CoreLib.dll reemplaza a mscorlib.dll del Framework de .NET. Aunque en la documentación oficial aún se hacen referencias a mscorlib.dll.

A un nivel superior se encuentran tipos adicionales que complementan la funcionalidad a nivel del CLR, proporcionando características como XML, JSON, redes y Consultas Integradas en el Lenguaje (LINQ). Estos constituyen la Biblioteca de Clases Base (BCL, por sus siglas en inglés). Por encima de esto se encuentran las capas de aplicación, que proporcionan API para desarrollar tipos específicos de aplicaciones, como aplicaciones web o aplicaciones de cliente enriquecidas.

En este capítulo, proporcionamos lo siguiente:

- Una visión general de la BCL (que cubrimos en el resto del libro)
- Un resumen de alto nivel de las capas de aplicación.

Qué hay de nuevo en .NET 6

En .NET 6, la Biblioteca de Clases Base (BCL) tiene numerosas características nuevas. En particular:

• Las nuevas estructuras DateOnly/TimeOnly capturan de manera limpia una fecha o una hora para escenarios como registrar una fecha de cumpleaños o una hora de alarma (consulte "DateOnly y TimeOnly" en la página 292).

• Una nueva clase estática BitOperations proporciona acceso a operaciones numéricas base-2 de bajo nivel (consulte "BitOperations" en la página 320).

• Se han agregado los siguientes nuevos métodos LINQ: Chunk, DistinctBy, UnionBy, IntersectBy, ExceptBy, MinBy y MaxBy (consulte el Capítulo 9). Además, Take ahora acepta una variable Range.

• La nueva API JsonNode proporciona un DOM de escritura fluida cuyos tipos no requieren liberación de recursos (consulte "JsonNode" en la página 551). Utf8JsonWriter también tiene un nuevo método WriteRawValue.

• La nueva clase RandomAccess proporciona operaciones de E/S de archivos eficientes y seguras en subprocesos.

• La nueva clase NullabilityInfoContext en System.Reflection consulta las anotaciones de nulabilidad (consulte "NullabilityContextInfo" en la página 794).

• La clase RandomNumberGenerator en System.Security.Cryptography ahora tiene un método estático GetBytes(int) que devuelve una matriz de bytes aleatorios en una sola operación. También hay nuevos métodos para simplificar el cifrado y descifrado (consulte "Encrypting in Memory" en la página 856).

• Un nuevo método ForEachAsync en la clase Parallel limita la concurrencia asincrónica (consulte "Parallel.ForEachAsync" en la página 882). Task también tiene ahora un método WaitAsync, que aplica un tiempo de espera a cualquier operación asincrónica, y hay un nuevo temporizador diseñado para funcionar con await (consulte "PeriodicTimer" en la página 902).

• La nueva clase NativeMemory proporciona un contenedor ligero alrededor de las operaciones de asignación de memoria de bajo nivel, como malloc.

.NET 6 también presenta numerosas mejoras de rendimiento en el tiempo de ejecución y un soporte mejorado para procesadores Windows ARM64 y Apple M1/M2.

En las capas de aplicación, el cambio más importante es la introducción de MAUI (Multiplatform App UI, principios de 2022), que reemplaza a Xamarin para el desarrollo multiplataforma de aplicaciones móviles. MAUI también admite el desarrollo de aplicaciones de escritorio multiplataforma para macOS y Windows, y se orienta hacia un CLR/BCL unificado de .NET 6. UWP también tiene un sucesor, Windows App SDK (con WinUI 3 como su capa de presentación), y ha surgido una nueva tecnología llamada Blazor Desktop para escribir aplicaciones de escritorio y móviles basadas en HTML.

.NET Standard

La gran cantidad de bibliotecas públicas disponibles en NuGet no serían tan valiosas si solo admitieran .NET 6. Al escribir una biblioteca, a menudo querrás admitir una variedad de plataformas y versiones del tiempo de ejecución. Para lograr ese objetivo sin crear una compilación separada para cada tiempo de ejecución, debes apuntar al denominador común más bajo.

Esto es relativamente fácil si solo deseas admitir los predecesores directos de .NET 6: por ejemplo, si apuntas a .NET Core 3.0, tu biblioteca se ejecutará en .NET Core 3.0, .NET Core 3.1 y .NET 5+.

La situación se vuelve más complicada si también deseas admitir .NET Framework o Xamarin. Esto se debe a que cada uno de estos tiempos de ejecución tiene un CLR y BCL con características superpuestas, ninguno de ellos es un subconjunto puro de los demás.

.NET Standard resuelve este problema al definir subconjuntos artificiales que funcionan en toda una gama de tiempos de ejecución heredados. Al apuntar a .NET Standard, puedes escribir fácilmente bibliotecas con un alcance extenso.



.NET Standard no es un tiempo de ejecución; es simplemente una especificación que describe un conjunto mínimo de funcionalidades (tipos y miembros) que garantizan la compatibilidad con un conjunto determinado de tiempos de ejecución. El concepto es similar a las interfaces de C#: .NET Standard es como una interfaz que los tipos concretos (tiempos de ejecución) pueden implementar.

.NET Standard 2.0

La versión más útil es .NET Standard 2.0. Una biblioteca que se dirige a .NET Standard 2.0 en lugar de un tiempo de ejecución específico se ejecutará sin modificaciones en la mayoría de los tiempos de ejecución modernos y heredados que todavía se utilizan hoy en día, incluyendo:

- .NET Core 2.0+ (incluyendo .NET 5 y .NET 6)
- UWP 10.0.16299+
- Mono 5.4+ (el CLR/BCL utilizado por versiones antiguas de Xamarin)
- .NET Framework 4.6.1+

Para apuntar a .NET Standard 2.0, agrega lo siguiente a tu archivo .csproj:

```
<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
<PropertyGroup>
```

La mayoría de las API descritas en este libro son compatibles con .NET Standard 2.0.

.NET Standard 2.1

.NET Standard 2.1 es un superconjunto de .NET Standard 2.0 que admite (solo) las siguientes plataformas:

- .NET Core 3+
- Mono 6.4+

.NET Standard 2.1 no es compatible con ninguna versión de .NET Framework (y tampoco con UWP), lo que lo hace mucho menos útil que .NET Standard 2.0.

Las siguientes API, en particular, están disponibles en .NET Standard 2.1 (pero no en .NET Standard 2.0):

- Span<T> (Capítulo 23)
- Reflection.Emit (Capítulo 18)
- ValueTask<T> (Capítulo 14)

.NET Standards más antiguos

También existen versiones más antiguas de .NET Standard, especialmente 1.1, 1.2, 1.3 y 1.6. Un estándar con un número más alto siempre es un superconjunto estricto de un estándar con un número más bajo. Por ejemplo, si escribes una biblioteca que apunta a .NET Standard 1.6, admitirás no solo las versiones recientes de los principales tiempos de ejecución, sino también .NET Core 1.0. Y si apuntas a .NET Standard 1.3, admitirás todo lo que ya hemos mencionado, además de .NET Framework 4.6.0. La siguiente tabla detalla esto:

If you target...	You also support...
Standard 1.6	.NET Core 1.0
Standard 1.3	Above plus .NET 4.6.0
Standard 1.2	Above plus .NET 4.5.1, Windows Phone 8.1, WinRT for Windows 8.1
Standard 1.1	Above plus .NET 4.5.0, Windows Phone 8.0, WinRT for Windows 8.0

 Los estándares 1.x carecen de miles de API que están presentes en 2.0, incluyendo gran parte de lo que describimos en este libro. Esto puede hacer que apuntar a un estándar 1.x sea significativamente más desafiante, especialmente si necesitas integrar código o bibliotecas existentes.

También puedes pensar en .NET Standard como un denominador común mínimo. En el caso de .NET Standard 2.0, los tiempos de ejecución que lo implementan tienen un BCL similar, por lo que el denominador común mínimo es grande y útil. Sin embargo, si también deseas compatibilidad con .NET Core 1.0 (con su BCL significativamente reducido), el denominador común mínimo, .NET Standard 1.x, se vuelve mucho más pequeño y menos útil.

Compatibilidad entre .NET Framework y .NET 6

Debido a que .NET Framework ha existido durante mucho tiempo, no es raro encontrar bibliotecas que solo están disponibles para .NET Framework (sin equivalente en .NET Standard, .NET Core o .NET 6). Para mitigar esta situación, los proyectos de .NET 5+ y .NET Core pueden hacer referencia a ensamblados de .NET Framework, con las siguientes condiciones:

- Se genera una excepción si el ensamblado de .NET Framework llama a una API no compatible.
- Las dependencias no triviales pueden no resolverse correctamente.

En la práctica, es más probable que esto funcione en casos simples, como un ensamblado que envuelve una DLL no administrada.

Versiones de tiempo de ejecución y lenguaje C#

Por defecto, la versión de tiempo de ejecución de tu proyecto determina la versión de lenguaje C# que se utiliza:

- Para .NET 6, es C# 10.
- Para .NET 5, es C# 9.
- Para .NET Core 3.x, Xamarin y .NET Standard 2.1, es C# 8.
- Para .NET Core 2.x, .NET Framework y .NET Standard 2.0 y versiones anteriores, es C# 7.3.

Esto se debe a que las versiones posteriores de C# dependen de tipos que solo están disponibles en versiones posteriores de tiempo de ejecución.

Puedes anular la versión de lenguaje en el archivo de proyecto con el elemento <LangVersion>. Sin embargo, utilizar una versión de lenguaje más reciente (como C# 10) con una versión de

tiempo de ejecución más antigua (como .NET 5) significa que las características del lenguaje que dependen de tipos más nuevos de .NET no funcionarán (aunque en algunos casos puedes definir esos tipos por ti mismo).

Ensamblados de referencia

Cuando apuntas a .NET Standard, tu proyecto hace referencia de manera implícita a un ensamblado llamado netstandard.dll, que contiene todos los tipos y miembros permitidos para la versión de .NET Standard seleccionada. Esto se llama ensamblado de referencia porque existe únicamente para beneficio del compilador y no contiene código compilado. En tiempo de ejecución, los "verdaderos" ensamblados se identifican a través de atributos de redirección de ensamblados (la elección de los ensamblados dependerá del tiempo de ejecución y plataforma en la que finalmente se ejecute el ensamblado).

Curiosamente, ocurre algo similar cuando apuntas a .NET 6. Tu proyecto hace referencia implícitamente a un conjunto de ensamblados de referencia cuyos tipos reflejan lo que está en los ensamblados de tiempo de ejecución para la versión de .NET elegida. Esto ayuda con la versionización y la compatibilidad entre plataformas, y también te permite apuntar a una versión de .NET diferente a la que está instalada en tu máquina.

El CLR y el BCL

Tipos del sistema

Los tipos más fundamentales se encuentran directamente en el espacio de nombres System. Estos incluyen los tipos incorporados de C#; la clase base Exception; las clases base Enum, Array y Delegate; y Nullable, Type, DateTime, TimeSpan y Guid. El espacio de nombres System también incluye tipos para realizar funciones matemáticas (Math), generar números aleatorios (Random) y convertir entre varios tipos (Convert y BitConverter).

El Capítulo 6 describe estos tipos, así como las interfaces que definen protocolos estándar utilizados en .NET para tareas como el formato (IFormattable) y la comparación de orden (IComparable).

El espacio de nombres System también define la interfaz IDisposable y la clase GC para interactuar con el recolector de basura, que se describe en el Capítulo 12.

Procesamiento de texto

El espacio de nombres System.Text contiene la clase StringBuilder (el equivalente editable o mutable de string) y los tipos para trabajar con codificaciones de texto, como UTF-8 (Encoding y sus subtipos). Esto se cubre en el Capítulo 6.

El espacio de nombres System.Text.RegularExpressions contiene tipos que realizan operaciones avanzadas de búsqueda y reemplazo basadas en patrones; estos se describen en el Capítulo 25.

Colecciones

.NET ofrece una variedad de clases para administrar colecciones de elementos. Estas incluyen estructuras basadas tanto en listas como en diccionarios; funcionan en conjunto con un conjunto de interfaces estándar que unifican sus características comunes. Todos los tipos de colecciones se definen en los siguientes espacios de nombres, que se cubren en el Capítulo 7:

```
System.Collections                // Nongeneric collections
System.Collections.Generic        // Generic collections
System.Collections.Specialized    // Strongly typed collections
System.Collections.ObjectModel    // Bases for your own collections
System.Collections.Concurrent     // Thread-safe collections (Chapter 22)
```

Consulta(Querying)

La consulta de lenguaje integrado (LINQ) te permite realizar consultas seguras en cuanto al tipo sobre colecciones locales y remotas (por ejemplo, tablas de SQL Server) y se describe en los capítulos 8 a 10. Una gran ventaja de LINQ es que presenta una API de consulta coherente en una variedad de dominios. Los tipos esenciales se encuentran en los siguientes espacios de nombres:

```
System.Linq                 // LINQ to Objects and PLINQ
System.Linq.Expressions     // For building expressions manually
System.Xml.Linq             // LINQ to XML
```

XML y JSON

XML y JSON son ampliamente compatibles en .NET. El capítulo 10 se centra por completo en LINQ to XML, un modelo de objeto de documento XML (DOM) ligero que se puede construir y consultar mediante LINQ. El capítulo 11 cubre las clases de bajo nivel de lectura/escritura XML eficiente, los esquemas y hojas de estilo XML y los tipos para trabajar con JSON:

```
System.Xml                // XmlReader, XmlWriter
System.Xml.Linq           // The LINQ to XML DOM
System.Xml.Schema         // Support for XSD
System.Xml.Serialization  // Declarative XML serialization for .NET types
System.Xml.XPath          // XPath query language
System.Xml.Xsl            // Stylesheet support

System.Text.Json          // JSON reader/writer and DOM
```

En el suplemento en línea en http://www.albahari.com/nutshell, se cubre el serializador JSON.

Diagnostics

En el capítulo 13, se cubre el registro y las aserciones, y se describe cómo interactuar con otros procesos, escribir en el registro de eventos de Windows y manejar el monitoreo de rendimiento. Los tipos para esto se definen en System.Diagnostics y sus subespacios de nombres.

Concurrencia y asincronía

Muchas aplicaciones modernas necesitan lidiar con más de una tarea que ocurre al mismo tiempo. Desde C# 5.0, esto se ha vuelto más fácil a través de funciones asíncronas y constructos de alto nivel como tareas y combinadores de tareas. El capítulo 14 explica todo esto en detalle,

después de comenzar con los conceptos básicos de la programación multiproceso. Los tipos para trabajar con subprocesos y operaciones asíncronas se encuentran en los espacios de nombres System.Threading y System.Threading.Tasks.

Flujos y E/S (Entrada/Salida)

.NET proporciona un modelo basado en flujos para la E/S de bajo nivel. Los flujos se utilizan típicamente para leer y escribir directamente en archivos y conexiones de red, y se pueden encadenar o envolver en flujos decoradores para agregar funcionalidad de compresión o cifrado. El capítulo 15 describe la arquitectura de los flujos, así como el soporte específico para trabajar con archivos y directorios, compresión, tuberías y archivos mapeados en memoria. Los tipos de flujos y E/S están definidos en el espacio de nombres System.IO, y los tipos de E/S de archivos para Windows Runtime (WinRT) se encuentran en el espacio de nombres Windows.Storage.

Redes

Puedes acceder directamente a la mayoría de los protocolos de red estándar, como HTTP, TCP/IP y SMTP, a través de los tipos en System.Net. En el capítulo 16, demostramos cómo comunicarse utilizando cada uno de estos protocolos, comenzando con tareas simples como descargar desde una página web y terminando con el uso directo de TCP/IP para recuperar correo POP3. Estos son los espacios de nombres que cubrimos:

```
System.Net
System.Net.Http          // HttpClient
System.Net.Mail          // For sending mail via SMTP
System.Net.Sockets       // TCP, UDP, and IP
```

Ensamblados, reflexión y atributos

Los ensamblados en los que se compilan los programas de C# incluyen instrucciones ejecutables (almacenadas como IL) y metadatos, que describen los tipos, miembros y atributos del programa. A través de la reflexión, puedes inspeccionar estos metadatos en tiempo de ejecución y hacer cosas como invocar métodos dinámicamente. Con Reflection.Emit, puedes construir código nuevo sobre la marcha.

En el capítulo 17, describimos la estructura de los ensamblados y cómo cargarlos y aislarlos dinámicamente. En el capítulo 18, cubrimos la reflexión y los atributos, describiendo cómo inspeccionar metadatos, invocar funciones dinámicamente, escribir atributos personalizados, generar nuevos tipos y analizar IL en bruto. Los tipos para utilizar la reflexión y trabajar con ensamblados se encuentran en los siguientes espacios de nombres:

```
System
System.Reflection
System.Reflection.Emit
```

Programación dinámica

En el capítulo 19, examinamos algunos de los patrones de programación dinámica y utilizamos el tiempo de ejecución de lenguajes dinámicos (DLR). Describimos cómo implementar el patrón Visitor, escribir objetos dinámicos personalizados e interoperar con IronPython. Los tipos para la programación dinámica se encuentran en System.Dynamic.

Cifrado

.NET proporciona un amplio soporte para protocolos populares de hash y cifrado. En el capítulo 20, cubrimos el hash, el cifrado simétrico y asimétrico, y la API de Protección de Datos de Windows. Los tipos para esto se definen en:

```
System.Security
System.Security.Cryptography
```

Programación de subprocesos avanzada

Las funciones asíncronas de C# hacen que la programación concurrente sea significativamente más fácil, ya que reducen la necesidad de técnicas de nivel inferior. Sin embargo, todavía hay momentos en los que se necesitan construcciones de señalización, almacenamiento local de subprocesos, bloqueos de lectura/escritura, y más. El capítulo 21 explica esto en profundidad. Los tipos de subprocesamiento están en el espacio de nombres System.Threading.

Programación paralela

En el capítulo 22, cubrimos en detalle las bibliotecas y tipos para aprovechar los procesadores multinúcleo, incluyendo APIs para el paralelismo de tareas, el paralelismo de datos imperativo y el paralelismo funcional (PLINQ).

Span<T> y Memory<T>

Para ayudar con la optimización de micro-optimización de puntos críticos de rendimiento, el CLR proporciona una serie de tipos que te ayudan a programar de tal manera que se reduce la carga en el administrador de memoria. Dos de los tipos clave son Span<T> y Memory<T>, que describimos en el capítulo 23.

Interoperabilidad nativa y COM

Puedes interoperar con código nativo y Component Object Model (COM). La interoperabilidad nativa te permite llamar funciones en DLL no administradas, registrar devoluciones de llamada, mapear estructuras de datos e interoperar con tipos de datos nativos. La interoperabilidad con COM te permite llamar tipos de COM (en máquinas Windows) y exponer tipos .NET a COM. Los tipos que admiten estas funciones se encuentran en System.Runtime.InteropServices, y los cubrimos en el capítulo 24.

Expresiones regulares

En el capítulo 25, cubrimos cómo puedes usar expresiones regulares para buscar patrones de caracteres en cadenas.

Serialización

.NET proporciona varios sistemas para guardar y restaurar objetos en una representación binaria o de texto. Estos sistemas se pueden utilizar para la comunicación, así como para guardar y restaurar objetos en un archivo. En el suplemento en línea en http://www.albahari.com/nutshell, cubrimos los cuatro motores de serialización: el serializador binario, el serializador JSON (recientemente actualizado), el serializador XML y el serializador de contrato de datos.

El compilador Roslyn

El compilador de C# en sí mismo está escrito en C# y el proyecto se llama "Roslyn". Las bibliotecas de Roslyn están disponibles como paquetes de NuGet. Con estas bibliotecas, puedes utilizar la funcionalidad del compilador de diversas formas además de compilar código fuente en un ensamblado, como escribir herramientas de análisis de código y refactorización. En la suplemento en línea, en http://www.albahari.com/nutshell, se cubre Roslyn.

Capas de aplicación

Las aplicaciones basadas en interfaz de usuario (UI) se pueden dividir en dos categorías: cliente ligero, que corresponde a un sitio web, y cliente rico, que es un programa que el usuario final debe descargar e instalar en una computadora o dispositivo móvil.

Para escribir aplicaciones de cliente ligero en C#, existe ASP.NET Core, que se ejecuta en Windows, Linux y macOS. ASP.NET Core también está diseñado para escribir API web.

Para aplicaciones de cliente rico, hay varias opciones de API:

- La capa de escritorio de Windows incluye las API de WPF y Windows Forms, y se ejecuta en escritorios con Windows 7/8/10/11.
- UWP te permite escribir aplicaciones de la Tienda Windows que se ejecutan en escritorios y dispositivos con Windows 10+.
- WinUI 3 (Windows App SDK) es un sucesor reciente de UWP que se ejecuta en escritorios con Windows 10+.
- MAUI (anteriormente Xamarin) se ejecuta en dispositivos móviles iOS y Android. MAUI también permite crear aplicaciones de escritorio multiplataforma que se dirigen a macOS y Windows.

También existen bibliotecas de terceros, como Avalonia, que ofrecen soporte de interfaz de usuario multiplataforma.

ASP.NET Core

ASP.NET Core es un sucesor modular y ligero de ASP.NET y es adecuado para crear sitios web, API web basadas en REST y microservicios. También puede ejecutarse junto con dos frameworks populares de aplicaciones de página única: React y Angular. ASP.NET admite el popular patrón Modelo-Vista-Controlador (MVC), así como una tecnología más nueva llamada Blazor, donde el código del lado del cliente se escribe en C# en lugar de JavaScript.

ASP.NET Core se ejecuta en Windows, Linux y macOS, y puede alojarse en un proceso personalizado. A diferencia de su predecesor .NET Framework (ASP.NET), ASP.NET Core no depende de System.Web ni de la carga histórica de los formularios web.

Como cualquier arquitectura de cliente delgado, ASP.NET Core ofrece las siguientes ventajas generales sobre los clientes ricos:

- No hay despliegue en el extremo del cliente.
- El cliente puede ejecutarse en cualquier plataforma que admita un navegador web.
- Las actualizaciones se despliegan fácilmente.

Windows Desktop

La capa de aplicaciones de Windows Desktop ofrece dos opciones de API de interfaz de usuario para escribir aplicaciones de cliente rico: WPF (Windows Presentation Foundation) y Windows Forms. Ambas APIs se ejecutan en Windows Desktop/Server 7 hasta 11.

WPF

WPF se introdujo en 2006 y se ha mejorado continuamente desde entonces. A diferencia de su predecesor, Windows Forms, WPF renderiza explícitamente los controles utilizando DirectX, con los siguientes beneficios:

- Admite gráficos sofisticados, como transformaciones arbitrarias, renderizado en 3D, multimedia y transparencia real. El diseño de apariencia se admite a través de estilos y plantillas.
- Su unidad de medida principal no se basa en píxeles, por lo que las aplicaciones se muestran correctamente en cualquier configuración de DPI.
- Tiene un amplio y flexible soporte de diseño, lo que significa que se puede localizar una aplicación sin el riesgo de que los elementos se superpongan.
- El uso de DirectX permite un renderizado rápido y la capacidad de aprovechar la aceleración de hardware gráfico.
- Ofrece una vinculación de datos confiable.
- Las interfaces de usuario se pueden describir de forma declarativa en archivos XAML que se pueden mantener de forma independiente de los archivos de "código detrás", lo que ayuda a separar la apariencia de la funcionalidad.

WPF lleva tiempo aprenderlo debido a su tamaño y complejidad. Los tipos para escribir aplicaciones de WPF se encuentran en el espacio de nombres System.Windows y en todos los subespacios de nombres, excepto System.Windows.Forms.

Windows Forms

Windows Forms es una API de cliente rico que se incluyó con la primera versión de .NET Framework en 2000. En comparación con WPF, Windows Forms es una tecnología relativamente sencilla que proporciona la mayoría de las características necesarias para escribir una aplicación de Windows típica. También tiene una relevancia significativa en el mantenimiento de aplicaciones heredadas. Sin embargo, en comparación con WPF, Windows Forms tiene numerosas desventajas, la mayoría de las cuales se derivan de ser un envoltorio sobre GDI+ y la biblioteca de controles Win32:

- Aunque proporciona mecanismos para la detección de DPI, sigue siendo demasiado fácil escribir aplicaciones que no funcionan en clientes cuya configuración de DPI difiere de la del desarrollador.
- La API para dibujar controles no estándar es GDI+, que, aunque es razonablemente flexible, es lenta para renderizar áreas grandes (y sin doble búfer, puede parpadear).
- Los controles carecen de transparencia real.
- La mayoría de los controles no son compositivos. Por ejemplo, no se puede colocar un control de imagen dentro del encabezado de un control de pestaña. Personalizar listas, cuadros combinados y controles de pestaña de una manera que sería trivial con WPF es lento y doloroso en Windows Forms.
- Es difícil implementar correctamente el diseño dinámico de forma fiable.

El último punto es una excelente razón para favorecer WPF sobre Windows Forms, incluso si estás escribiendo una aplicación empresarial que solo necesita una interfaz de usuario y no una "experiencia de usuario". Los elementos de diseño en WPF, como Grid, facilitan el ensamblaje de etiquetas y cuadros de texto de manera que siempre se alineen, incluso después de la localización que cambia el idioma, sin lógica complicada y sin parpadeo. Además, no necesitas ceder ante el denominador común más bajo en cuanto a resolución de pantalla; los elementos de diseño de WPF se han diseñado desde el principio para adaptarse correctamente al redimensionamiento.

En el lado positivo, Windows Forms es relativamente fácil de aprender y aún cuenta con un buen número de controles de terceros.

Los tipos de Windows Forms se encuentran en los espacios de nombres System.Windows.Forms (en System.Windows.Forms.dll) y System.Drawing (en System.Drawing.dll). Este último también contiene los tipos de GDI+ para dibujar controles personalizados.

UWP y WinUI 3

UWP es una API de cliente rico para escribir interfaces de usuario diseñadas para dispositivos táctiles que se dirigen a escritorios y dispositivos con Windows 10 o superior. La palabra "Universal" se refiere a su capacidad para ejecutarse en una variedad de dispositivos con Windows 10, incluidos Xbox, Surface Hub y HoloLens. Sin embargo, no es compatible con versiones anteriores de Windows, como Windows 7 y Windows 8/8.1.

La API UWP utiliza XAML y es algo similar a WPF. Estas son sus principales diferencias:

- El modo principal de distribución para las aplicaciones UWP es la Tienda de Windows.
- Las aplicaciones UWP se ejecutan en un entorno seguro para reducir la amenaza de malware, lo que significa que no pueden realizar tareas como leer o escribir archivos arbitrarios, y no pueden ejecutarse con elevación administrativa.
- UWP se basa en tipos WinRT que forman parte del sistema operativo (Windows), no del tiempo de ejecución administrado. Esto significa que al escribir aplicaciones, debes especificar un rango de versiones de Windows (como Windows 10 compilación 17763 a Windows 10 compilación 18362). Esto implica que debes orientar una API antigua o requerir que tus clientes instalen la última actualización de Windows.

Debido a las limitaciones creadas por estas diferencias, UWP nunca logró igualar la popularidad de WPF y Windows Forms. Para abordar esto, Microsoft ha introducido una nueva tecnología para reemplazar a UWP llamada Windows App SDK (con una capa de interfaz de usuario llamada WinUI 3).

El Windows App SDK traslada las API de WinRT desde el sistema operativo al tiempo de ejecución, lo que expone una interfaz completamente administrada y elimina la necesidad de orientar un rango de versiones específico del sistema operativo. Además, hace lo siguiente:

- Se integra mejor con las API de Windows Desktop (Windows Forms y WPF).
- Te permite escribir aplicaciones que se ejecutan fuera del entorno seguro de la Tienda de Windows.
- Se ejecuta sobre la última versión de .NET (en lugar de estar vinculado a .NET Core 2.2, como ocurre con UWP).

Sin embargo, en el momento de escribir esto, el Windows App SDK no es compatible con Xbox ni HoloLens.

MAUI

MAUI (a principios de 2022) es la nueva encarnación de Xamarin y te permite desarrollar aplicaciones móviles en C# que se orientan a iOS y Android (así como aplicaciones de escritorio multiplataforma que se orientan a macOS y Windows).

El CLR/BCL que se ejecuta en iOS y Android se llama Mono (una derivación del tiempo de ejecución Mono de código abierto). Históricamente, Mono no ha sido completamente compatible con .NET, y las bibliotecas que se ejecutaban en Mono y .NET se orientaban a .NET Standard. Con MAUI, la interfaz pública de Mono se ha fusionado con .NET 6, lo que hace que Mono sea, en efecto, una implementación de .NET 6.

Las características nuevas de MAUI incluyen una interfaz de proyecto unificada, recarga en caliente, compatibilidad con Blazor Desktop y aplicaciones híbridas, y mejoras en rendimiento y tiempos de inicio. Consulta https://github.com/dotnet/maui para obtener más información.

6 Fundamentos de .NET

Muchas de las funcionalidades principales que necesitas al programar no son proporcionadas por el lenguaje C#, sino por los tipos en la Biblioteca de Clases Base (.NET BCL, por sus siglas en inglés). En este capítulo, cubrimos tipos que ayudan con tareas fundamentales de programación, como la comparación de igualdad virtual, la comparación de orden y la conversión de tipos. También cubrimos los tipos básicos de .NET, como String, DateTime y Enum.

Los tipos en esta sección se encuentran en el espacio de nombres System, con las siguientes excepciones:

- StringBuilder se define en System.Text, al igual que los tipos para codificaciones de texto.
- CultureInfo y los tipos asociados se definen en System.Globalization.
- XmlConvert se define en System.Xml.

Manipulación de cadenas y texto

Char

Un char en C# representa un único carácter Unicode y es un alias de la estructura System.Char. En el Capítulo 2, describimos cómo expresar literales de char:

```
char c = 'A';
char newLine = '\n';
```

System.Char define una serie de métodos estáticos para trabajar con caracteres, como ToUpper, ToLower e IsWhiteSpace. Puedes llamar a estos métodos tanto a través del tipo System.Char como mediante su alias char:

```
Console.WriteLine (System.Char.ToUpper ('c'));    // C
Console.WriteLine (char.IsWhiteSpace ('\t'));     // True
```

ToUpper y ToLower respetan la configuración regional del usuario final, lo cual puede dar lugar a errores sutiles. La siguiente expresión evalúa a false en Turquía:

```
char.ToUpper ('i') == 'I'
```

La razón de esto es que en Turquía, char.ToUpper ('i') es 'İ' (¡observa el punto encima!). Para evitar este problema, System.Char (y System.String) también proporciona versiones invariantes de cultura de ToUpper y ToLower que terminan con la palabra Invariant. Estas siempre aplican las reglas de la cultura inglesa:

```
Console.WriteLine (char.ToUpperInvariant ('i'));    // I
```

Esto es un atajo para:

```
Console.WriteLine (char.ToUpper ('i', CultureInfo.InvariantCulture))
```

Para obtener más información sobre regiones y culturas, consulta "Formateo y análisis" en la página 297.

La mayoría de los métodos estáticos restantes de char están relacionados con la categorización de caracteres. La Tabla 6-1 enumera estos métodos.

Table 6-1. Static methods for categorizing characters

Static method	Characters included	Unicode categories included

`IsLetter`	A–Z, a–z, and letters of other alphabets	`UpperCaseLetter`
		`LowerCaseLetter`
		`TitleCaseLetter`
		`ModifierLetter`
		`OtherLetter`

`IsUpper`	Uppercase letters	`UpperCaseLetter`
`IsLower`	Lowercase letters	`LowerCaseLetter`
`IsDigit`	0–9 plus digits of other alphabets	`DecimalDigitNumber`
`IsLetterOrDigit`	Letters plus digits	(`IsLetter`, `IsDigit`)

`IsNumber`	All digits plus Unicode fractions and Roman numeral symbols	`DecimalDigitNumber`
		`LetterNumber`
		`OtherNumber`

`IsSeparator`	Space plus all Unicode separator characters	`LineSeparator` `ParagraphSeparator`
`IsWhiteSpace`	All separators plus `\n`, `\r`, `\t`, `\f`, and `\v`	`LineSeparator` `ParagraphSeparator`

`IsPunctuation`	Symbols used for punctuation in Western and other alphabets	`DashPunctuation`
		`ConnectorPunctuation`
		`InitialQuotePunctuation`
		`FinalQuotePunctuation`

`IsSymbol`	Most other printable symbols	`MathSymbol`
		`ModifierSymbol`
		`OtherSymbol`

`IsControl`	Nonprintable "control" characters below 0x20, such as `\r`, `\n`, `\t`, `\0`, and characters between 0x7F and 0x9A	(None)

Para una categorización más detallada, char proporciona un método estático llamado GetUnicodeCategory; este devuelve una enumeración UnicodeCategory cuyos miembros se muestran en la columna más a la derecha de la Tabla 6-1.



Al realizar una conversión explícita desde un entero, es posible generar un char fuera del conjunto Unicode asignado. Para comprobar la validez de un carácter, llama a char.GetUnicodeCategory: si el resultado es UnicodeCategory.OtherNotAssigned, el carácter es inválido.

Un char tiene una longitud de 16 bits, lo cual es suficiente para representar cualquier carácter Unicode en el Plano Multilingüe Básico. Para ir más allá de esto, debes utilizar pares de

sustitutos: describimos los métodos para hacer esto en "Codificaciones de texto y Unicode" en la página 281.

String

Un string en C# (== System.String) es una secuencia inmutable (no modificable) de caracteres. En el Capítulo 2, describimos cómo expresar literales de string, realizar comparaciones de igualdad y concatenar dos strings. Esta sección cubre las funciones restantes para trabajar con strings, expuestas a través de los miembros estáticos e instanciados de la clase System.String.

Creación de strings

La forma más sencilla de crear un string es asignando un literal, como vimos en el Capítulo 2:

```
string s1 = "Hello";
string s2 = "First Line\r\nSecond Line";
string s3 = @"\\server\fileshare\helloworld.cs";
```

Para crear una secuencia repetitiva de caracteres, puedes utilizar el constructor de string:

```
Console.Write (new string ('*', 10));      // **********
```

También puedes construir un string a partir de un arreglo de chars. El método ToCharArray realiza la operación inversa:

```
char[] ca = "Hello".ToCharArray();
string s = new string (ca);              // s = "Hello"
```

El constructor de string también tiene sobrecargas para aceptar varios tipos de punteros (inseguros), con el fin de crear strings a partir de tipos como char*.

Strings nulos y vacíos

Un string vacío tiene una longitud de cero. Para crear un string vacío, puedes utilizar tanto un literal como el campo estático string.Empty; para comprobar si un string está vacío, puedes realizar una comparación de igualdad o comprobar su propiedad Length:

```
string empty = "";
Console.WriteLine (empty == "");              // True
Console.WriteLine (empty == string.Empty);    // True
Console.WriteLine (empty.Length == 0);        // True
```

Debido a que los strings son tipos de referencia, también pueden ser nulos:

```
string nullString = null;
Console.WriteLine (nullString == null);        // True
Console.WriteLine (nullString == "");          // False
Console.WriteLine (nullString.Length == 0);    // NullReferenceException
```

El método estático string.IsNullOrEmpty es un atajo útil para comprobar si un string dado es nulo o vacío.

Acceso a caracteres dentro de un string

El indexador de un string devuelve un solo carácter en el índice dado. Al igual que con todas las funciones que operan en strings, esto se indexa desde cero:

```
string str  = "abcde";
char letter = str[1];        // letter == 'b'
```

Además, el string implementa IEnumerable<char>, por lo que puedes recorrer sus caracteres con foreach:

```
foreach (char c in "123") Console.Write (c + ",");    // 1,2,3,
```

Búsqueda dentro de strings

Los métodos más simples para buscar dentro de strings son StartsWith, EndsWith y Contains. Todos ellos devuelven true o false:

```
Console.WriteLine ("quick brown fox".EndsWith ("fox"));      // True
Console.WriteLine ("quick brown fox".Contains ("brown"));    // True
```

StartsWith y EndsWith tienen sobrecargas que te permiten especificar un enum StringComparison o un objeto CultureInfo para controlar la sensibilidad a mayúsculas y minúsculas y la cultura (ver "Comparación ordinal versus cultura" en la página 278). El valor predeterminado es realizar una coincidencia que distingue entre mayúsculas y minúsculas utilizando las reglas aplicables a la cultura actual (localizada). El siguiente ejemplo realiza una búsqueda sin distinción entre mayúsculas y minúsculas utilizando las reglas de la cultura invariante:

```
"abcdef".StartsWith ("aBc", StringComparison.InvariantCultureIgnoreCase)
```

El método Contains no ofrece la conveniencia de esta sobrecarga, aunque se puede lograr el mismo resultado con el método IndexOf.

IndexOf es más poderoso: devuelve la primera posición de un carácter o subcadena dada (o -1 si no se encuentra la subcadena):

```
Console.WriteLine ("abcde".IndexOf ("cd"));   // 2
```

IndexOf también tiene sobrecargas para aceptar una startPosition (un índice desde el cual comenzar la búsqueda) y un enum StringComparison:

```
Console.WriteLine ("abcde abcde".IndexOf ("CD", 6,
                   StringComparison.CurrentCultureIgnoreCase));    // 8
```

LastIndexOf es similar a IndexOf, pero funciona en sentido inverso a través del string.

IndexOfAny devuelve la primera posición que coincide con cualquiera de los caracteres de un conjunto:

```
Console.Write ("ab,cd ef".IndexOfAny (new char[] {' ', ','} ));       // 2
Console.Write ("pas5w0rd".IndexOfAny ("0123456789".ToCharArray() ));  // 3
```

LastIndexOfAny hace lo mismo en dirección inversa.

Manipulación de strings

Dado que String es inmutable, todos los métodos que "manipulan" un string devuelven uno nuevo, dejando el original sin cambios (lo mismo ocurre cuando reasignas una variable de string).

Substring extrae una porción de un string:

```
string left3 = "12345".Substring (0, 3);     // left3 = "123";
string mid3  = "12345".Substring (1, 3);     // mid3 = "234";
```

Si omites la longitud, obtienes el resto del string:

```
string end3  = "12345".Substring (2);        // end3 = "345";
```

Insert y Remove insertan o eliminan caracteres en una posición especificada:

```
string s1 = "helloworld".Insert (5, ", ");    // s1 = "hello, world"
string s2 = s1.Remove (5, 2);                 // s2 = "helloworld";
```

PadLeft y PadRight rellenan un string hasta una longitud determinada con un carácter especificado (o un espacio si no se especifica):

```
Console.WriteLine ("12345".PadLeft (9, '*'));  // ****12345
Console.WriteLine ("12345".PadLeft (9));       //     12345
```

Si el string de entrada es más largo que la longitud de relleno, se devuelve el string original sin cambios.

TrimStart y TrimEnd eliminan caracteres especificados del principio o final de un string; Trim hace ambas operaciones. De forma predeterminada, estas funciones eliminan los caracteres de espacio en blanco (incluyendo espacios, tabulaciones, saltos de línea y variaciones Unicode de estos):

```
Console.WriteLine ("  abc \t\r\n ".Trim().Length);   // 3
```

Replace reemplaza todas las ocurrencias (no superpuestas) de un carácter o subcadena en particular:

```
Console.WriteLine ("to be done".Replace (" ", " | ") );  // to | be | done
Console.WriteLine ("to be done".Replace (" ", "")    );  // tobedone
```

ToUpper y ToLower devuelven versiones en mayúsculas y minúsculas del string de entrada. De forma predeterminada, respetan la configuración de idioma actual del usuario; ToUpperInvariant y ToLowerInvariant siempre aplican las reglas del alfabeto inglés.

Dividir y unir strings

Split divide un string en piezas:

```
string[] words = "The quick brown fox".Split();

foreach (string word in words)
  Console.Write (word + "|");    // The|quick|brown|fox|
```

De forma predeterminada, Split utiliza caracteres de espacio en blanco como delimitadores; también se sobrecarga para aceptar una matriz de caracteres o strings como delimitadores. Split

también acepta opcionalmente un enum StringSplitOptions, que tiene una opción para eliminar las entradas vacías: esto es útil cuando las palabras están separadas por varios delimitadores seguidos.

El método estático Join realiza lo contrario que Split. Requiere un delimitador y una matriz de strings:

```
string[] words = "The quick brown fox".Split();
string together = string.Join (" ", words);      // The quick brown fox
```

El método estático Concat es similar a Join pero solo acepta una matriz de strings como parámetro y no aplica ningún separador. Concat es exactamente equivalente al operador + (de hecho, el compilador traduce + a Concat):

```
string sentence     = string.Concat ("The", " quick", " brown", " fox");
string sameSentence = "The" + " quick" + " brown" + " fox";
```

String.Format y formato compuesto de strings

El método estático Format proporciona una forma conveniente de construir strings que incluyen variables. Las variables incrustadas (o valores) pueden ser de cualquier tipo; el método Format simplemente llama a ToString en ellas.

El string maestro que incluye las variables incrustadas se llama formato compuesto. Al llamar a string.Format, se proporciona un string de formato compuesto seguido de cada una de las variables incrustadas:

```
string composite = "It's {0} degrees in {1} on this {2} morning";
string s = string.Format (composite, 35, "Perth", DateTime.Now.DayOfWeek);

// s == "It's 35 degrees in Perth on this Friday morning"
```

(Y eso es en grados Celsius)

Podemos usar literales de strings interpolados para lograr el mismo efecto (consultar "Tipo String" en la página 52). Solo se debe preceder el string con el símbolo $ y colocar las expresiones entre llaves:

```
string s = $"It's hot this {DateTime.Now.DayOfWeek} morning";
```

Cada número entre llaves se llama elemento de formato. El número corresponde a la posición del argumento y es seguido opcionalmente por:

- Una coma y un ancho mínimo para aplicar
- Dos puntos y una cadena de formato

El ancho mínimo es útil para alinear columnas. Si el valor es negativo, los datos se alinean a la izquierda; de lo contrario, se alinean a la derecha:

```
string composite = "Name={0,-20} Credit Limit={1,15:C}";

Console.WriteLine (string.Format (composite, "Mary", 500));
Console.WriteLine (string.Format (composite, "Elizabeth", 20000));
```

Aquí está el resultado:

```
Name=Mary                Credit Limit=        $500.00
Name=Elizabeth           Credit Limit=     $20,000.00
```

Aquí tienes el equivalente sin usar string.Format:

```
string s = "Name=" + "Mary".PadRight (20) +
           " Credit Limit=" + 500.ToString ("C").PadLeft (15);
```

El límite de crédito se formatea como moneda gracias a la cadena de formato "C". Describimos las cadenas de formato en detalle en "Formato y análisis" en la página 297.

Comparación de cadenas

Al comparar dos valores, .NET diferencia los conceptos de comparación de igualdad y comparación de orden. La comparación de igualdad prueba si dos instancias son semánticamente iguales; la comparación de orden determina cuál de las dos instancias (si alguna) viene primero al ordenarlas en secuencia ascendente o descendente.


La comparación de igualdad no es un subconjunto de la comparación de orden; los dos sistemas tienen diferentes propósitos. Es válido, por ejemplo, tener dos valores desiguales en la misma posición de ordenamiento. Retomamos este tema en "Comparación de igualdad" en la página 324.

Para la comparación de igualdad de cadenas, puedes usar el operador == o uno de los métodos Equals de la clase string. Estos últimos son más versátiles porque te permiten especificar opciones como la insensibilidad a mayúsculas y minúsculas.


Otra diferencia es que el operador == no funciona de manera confiable en cadenas si las variables se convierten al tipo object. Explicamos por qué esto ocurre en "Comparación de igualdad" en la página 324.

Para la comparación de orden de cadenas, puedes utilizar el método de instancia CompareTo o los métodos estáticos Compare y CompareOrdinal. Estos devuelven un número positivo o negativo, o cero, dependiendo de si el primer valor viene después, antes o junto al segundo.

Antes de entrar en los detalles de cada uno, debemos examinar los algoritmos subyacentes de comparación de cadenas en .NET.

Comparación ordinal versus comparación cultural

Existen dos algoritmos básicos para la comparación de cadenas: la comparación ordinal y la comparación sensible a la cultura. La comparación ordinal interpreta los caracteres simplemente como números (según su valor numérico Unicode), mientras que la comparación sensible a la cultura interpreta los caracteres en relación con un alfabeto específico. Existen dos culturas especiales: la "cultura actual", que se basa en la configuración seleccionada en el panel de

control de la computadora, y la "cultura invariante", que es la misma en todas las computadoras (y se asemeja a la cultura estadounidense).

Para la comparación de igualdad, tanto los algoritmos ordinal como los sensibles a la cultura son útiles. Sin embargo, para la ordenación, la comparación sensible a la cultura es casi siempre preferible. Para ordenar cadenas alfabéticamente, se necesita un alfabeto. La comparación ordinal se basa en los valores numéricos de los puntos Unicode, que casualmente colocan los caracteres del inglés en orden alfabético, pero no exactamente como se podría esperar. Por ejemplo, consideremos las cadenas "Atom", "atom" y "Zamia", asumiendo sensibilidad a mayúsculas. La cultura invariante las ordena de la siguiente manera:

```
"atom", "Atom", "Zamia"
```

La comparación ordinal, en cambio, las ordena de la siguiente manera:

```
"Atom", "Zamia", "atom"
```

Esto se debe a que la cultura invariante encapsula un alfabeto que considera que los caracteres en mayúsculas están junto a sus contrapartes en minúsculas (aAbBcCdD...). El algoritmo ordinal, sin embargo, coloca todos los caracteres en mayúsculas primero y luego todos los caracteres en minúsculas (A...Z, a...z). Esto es básicamente un vestigio del conjunto de caracteres ASCII inventado en la década de 1960.

Comparación de igualdad de cadenas

A pesar de las limitaciones de la comparación ordinal, el operador == de la clase string siempre realiza una comparación ordinal con distinción entre mayúsculas y minúsculas. Lo mismo ocurre con la versión de instancia del método string.Equals cuando se llama sin argumentos; esto define el comportamiento de comparación de igualdad "predeterminado" para el tipo string.



El algoritmo ordinal se eligió para los operadores == y Equals de la clase string debido a que es altamente eficiente y determinista. La comparación de igualdad de cadenas se considera fundamental y se realiza con mucha más frecuencia que la comparación de orden.

Una noción "estricta" de igualdad también es coherente con el uso general del operador ==.

A continuación, se presentan los siguientes métodos que permiten realizar comparaciones sensibles a la cultura o insensibles a mayúsculas y minúsculas:

```
public bool Equals(string value, StringComparison comparisonType);

public static bool Equals (string a, string b,
                           StringComparison comparisonType);
```

La versión estática de los métodos de comparación es ventajosa ya que sigue funcionando si una o ambas cadenas son nulas. StringComparison es un enumerado definido de la siguiente manera:

```
public enum StringComparison
{
  CurrentCulture,              // Case-sensitive
  CurrentCultureIgnoreCase,
  InvariantCulture,            // Case-sensitive
  InvariantCultureIgnoreCase,
  Ordinal,                     // Case-sensitive
  OrdinalIgnoreCase
}
```

Por ejemplo:

```
Console.WriteLine (string.Equals ("foo", "FOO",
                   StringComparison.OrdinalIgnoreCase));   // True

Console.WriteLine ("ǔ" == "ǔ");                            // False

Console.WriteLine (string.Equals ("Ǔ", "ǔ",
                   StringComparison.CurrentCulture));      // ?
```

(El resultado del tercer ejemplo depende de la configuración de idioma actual de la computadora).

Comparación de orden de cadenas

Para la comparación de orden de cadenas, el método de instancia CompareTo de la clase string realiza una comparación de orden sensible a la cultura y distingue entre mayúsculas y minúsculas. A diferencia del operador ==, CompareTo no utiliza una comparación ordinal. El método CompareTo tiene la siguiente definición:

```
public int CompareTo (string strB);
```



El método CompareTo implementa la interfaz genérica IComparable, que es un protocolo estándar de comparación utilizado en las bibliotecas .NET. Esto significa que CompareTo define el comportamiento de ordenación predeterminado de las cadenas en aplicaciones como colecciones ordenadas. Para obtener más información sobre IComparable, consulta "Comparación de orden" en la página 335.

Para otros tipos de comparación, puedes llamar a los métodos estáticos Compare y CompareOrdinal:

```
public static int Compare (string strA, string strB,
                           StringComparison comparisonType);

public static int Compare (string strA, string strB, bool ignoreCase,
                           CultureInfo culture);

public static int Compare (string strA, string strB, bool ignoreCase);

public static int CompareOrdinal (string strA, string strB);
```

Los dos últimos métodos son simplemente atajos para llamar a los primeros dos métodos.

Todos los métodos de comparación de orden devuelven un número positivo, un número negativo o cero, dependiendo de si el primer valor viene después, antes o junto al segundo valor:

```
Console.WriteLine ("Boston".CompareTo ("Austin"));    // 1
Console.WriteLine ("Boston".CompareTo ("Boston"));    // 0
Console.WriteLine ("Boston".CompareTo ("Chicago"));   // -1
Console.WriteLine ("Ǔ".CompareTo ("ǔ"));              // 0
Console.WriteLine ("foo".CompareTo ("FOO"));          // -1
```

El siguiente ejemplo realiza una comparación que no distingue entre mayúsculas y minúsculas utilizando la cultura actual:

```
Console.WriteLine (string.Compare ("foo", "FOO", true));   // 0
```

Al proporcionar un objeto CultureInfo, puedes usar cualquier alfabeto:

```
// CultureInfo is defined in the System.Globalization namespace

CultureInfo german = CultureInfo.GetCultureInfo ("de-DE");
int i = string.Compare ("Müller", "Muller", false, german);
```

StringBuilder

La clase StringBuilder (espacio de nombres System.Text) representa una cadena mutable (editable). Con StringBuilder, puedes agregar, insertar, eliminar y reemplazar subcadenas sin reemplazar todo el StringBuilder.

El constructor de StringBuilder opcionalmente acepta un valor de cadena inicial y un tamaño de capacidad interna inicial (el valor predeterminado es de 16 caracteres). Si te excedes de esto, StringBuilder redimensiona automáticamente sus estructuras internas para acomodar (a costa de un ligero costo de rendimiento) hasta su capacidad máxima (el valor predeterminado es int.MaxValue).

Un uso popular de StringBuilder es construir una cadena larga llamando repetidamente a Append. Este enfoque es mucho más eficiente que concatenar repetidamente tipos de cadenas ordinarios:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 50; i++) sb.Append(i).Append(",");
```

Para obtener el resultado final, llama a ToString():

```
Console.WriteLine (sb.ToString());

0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
```

El método AppendLine realiza un Append que agrega una secuencia de nueva línea ("\r\n" en Windows). El método AppendFormat acepta una cadena de formato compuesta, al igual que String.Format. Además de los métodos Insert, Remove y Replace (Replace funciona como Replace en las cadenas), StringBuilder define una propiedad Length y un índice modificable para obtener o establecer caracteres individuales.

Para borrar el contenido de un StringBuilder, puedes crear una nueva instancia o establecer su propiedad Length en cero.



Establecer la propiedad Length de un StringBuilder en cero no reduce su capacidad interna. Por lo tanto, si el StringBuilder anteriormente contenía un millón de caracteres, seguirá ocupando aproximadamente dos megabytes de memoria después de establecer su Length en cero. Si deseas liberar la memoria, debes crear un nuevo StringBuilder y permitir que el anterior quede fuera de alcance (y sea recolectado por el recolector de basura).

Codificaciones de texto y Unicode

Un conjunto de caracteres es una asignación de caracteres, cada uno con un código numérico o punto de código. Hay dos conjuntos de caracteres de uso común: Unicode y ASCII. Unicode tiene un espacio de direcciones de aproximadamente un millón de caracteres, de los cuales se han asignado aproximadamente 100,000. Unicode abarca la mayoría de los idiomas hablados en el mundo, así como algunos idiomas históricos y símbolos especiales. El conjunto ASCII es simplemente los primeros 128 caracteres del conjunto Unicode, que cubre la mayoría de lo que se ve en un teclado de estilo estadounidense. ASCII precede a Unicode por 30 años y aún se utiliza a veces por su simplicidad y eficiencia: cada carácter se representa por un byte.

El sistema de tipos de .NET está diseñado para funcionar con el conjunto de caracteres Unicode. Sin embargo, ASCII también es compatible de forma implícita al ser un subconjunto de Unicode.

Una codificación de texto mapea caracteres desde su punto de código numérico a una representación binaria. En .NET, las codificaciones de texto entran en juego principalmente al trabajar con archivos o flujos de texto. Cuando lees un archivo de texto en una cadena, un codificador de texto traduce los datos del archivo de binario a la representación interna de Unicode que esperan los tipos char y string. Una codificación de texto puede restringir qué caracteres se pueden representar y también puede afectar la eficiencia de almacenamiento.

Existen dos categorías de codificación de texto en .NET:

- Aquellas que mapean caracteres Unicode a otro conjunto de caracteres.
- Aquellas que utilizan esquemas de codificación Unicode estándar.

La primera categoría contiene codificaciones heredadas como el EBCDIC de IBM y conjuntos de caracteres de 8 bits con caracteres extendidos en la región de los primeros 128 caracteres, que eran populares antes de Unicode (identificados por una página de códigos). La codificación ASCII también está en esta categoría: codifica los primeros 128 caracteres y descarta todo lo demás. Esta categoría también incluye el conjunto GB18030 sin legado, que es el estándar obligatorio para aplicaciones escritas en China o vendidas en China desde el año 2000.

En la segunda categoría se encuentran UTF-8, UTF-16 y UTF-32 (y el obsoleto UTF-7). Cada uno difiere en eficiencia de espacio. UTF-8 es el más eficiente en cuanto al espacio para la mayoría de tipos de texto: utiliza entre uno y cuatro bytes para representar cada carácter. Los primeros 128 caracteres requieren solo un byte, lo que lo hace compatible con ASCII. UTF-8 es el encoding

más popular para archivos y flujos de texto (especialmente en internet) y es el valor predeterminado para la entrada/salida (I/O) de flujos en .NET (de hecho, es el valor predeterminado para casi todo lo que usa un encoding implícitamente).

UTF-16 utiliza una o dos palabras de 16 bits para representar cada carácter. Esto es lo que .NET utiliza internamente para representar caracteres y cadenas. Algunos programas también escriben archivos en UTF-16.

UTF-32 es el menos eficiente en cuanto al espacio: asigna directamente cada punto de código a 32 bits, por lo que cada carácter consume cuatro bytes. UTF-32 se usa raramente debido a esta razón. Sin embargo, facilita el acceso aleatorio porque cada carácter ocupa el mismo número de bytes.

Obteniendo un objeto Encoding

Para obtener un objeto Encoding, se utiliza la clase Encoding en el espacio de nombres System.Text. Hay varias subclases, cuyo propósito es encapsular familias de encodings con características similares. Los encodings más comunes se pueden obtener a través de propiedades estáticas dedicadas en la clase Encoding:

Encoding name	Static property on Encoding
UTF-8	`Encoding.UTF8`
UTF-16	`Encoding.Unicode` (*not* UTF16)
UTF-32	`Encoding.UTF32`
ASCII	`Encoding.ASCII`

Puedes obtener otros codificaciones llamando a Encoding.GetEncoding con un nombre de conjunto de caracteres estándar de la Autoridad de Números Asignados de Internet (IANA):

// En .NET 5+ y .NET Core, primero debes llamar a RegisterProvider:

```
Encoding.RegisterProvider (CodePagesEncodingProvider.Instance);

Encoding chinese = Encoding.GetEncoding ("GB18030");
```

El método estático GetEncodings devuelve una lista de todas las codificaciones admitidas junto con sus nombres estándar de IANA:

```
foreach (EncodingInfo info in Encoding.GetEncodings())
  Console.WriteLine (info.Name);
```

La otra forma de obtener una codificación es instanciar directamente una clase de codificación. Al hacerlo, puedes establecer varias opciones a través de argumentos de constructor, incluyendo:

- Si se debe lanzar una excepción si se encuentra una secuencia de bytes no válida al decodificar. El valor predeterminado es falso.

- Si se debe codificar/decodificar UTF-16/UTF-32 con los bytes más significativos primero (big endian) o los bytes menos significativos primero (little endian). El valor predeterminado es little endian, que es el estándar en el sistema operativo Windows.
- Si se debe emitir una marca de orden de bytes (byte-order mark) que indica el orden de bytes (endianness).

Codificación para E/S de archivos y flujos

La aplicación más común para un objeto Encoding es controlar cómo se lee y se escribe texto en un archivo o flujo. Por ejemplo, lo siguiente escribe "Testing..." en un archivo llamado data.txt con codificación UTF-16:

```
System.IO.File.WriteAllText ("data.txt", "Testing...", Encoding.Unicode);
```

Si omites el argumento final, WriteAllText aplica la ubicua codificación UTF-8.



UTF-8 es la codificación de texto predeterminada para todas las operaciones de entrada/salida de archivos y flujos.

Continuamos este tema en el Capítulo 15, en "Adaptadores de flujos" en la página 687.

Codificación a matrices de bytes

También puedes utilizar un objeto Encoding para convertir de y hacia una matriz de bytes. El método GetBytes convierte de string a byte[] con la codificación especificada; GetString convierte de byte[] a string:

```
byte[] utf8Bytes  = System.Text.Encoding.UTF8.GetBytes    ("0123456789");
byte[] utf16Bytes = System.Text.Encoding.Unicode.GetBytes ("0123456789");
byte[] utf32Bytes = System.Text.Encoding.UTF32.GetBytes   ("0123456789");

Console.WriteLine (utf8Bytes.Length);    // 10
Console.WriteLine (utf16Bytes.Length);   // 20
Console.WriteLine (utf32Bytes.Length);   // 40

string original1 = System.Text.Encoding.UTF8.GetString    (utf8Bytes);
string original2 = System.Text.Encoding.Unicode.GetString (utf16Bytes);
string original3 = System.Text.Encoding.UTF32.GetString   (utf32Bytes);

Console.WriteLine (original1);           // 0123456789
Console.WriteLine (original2);           // 0123456789
Console.WriteLine (original3);           // 0123456789
```

UTF-16 y pares de sustitutos

Recuerda que .NET almacena caracteres y cadenas en UTF-16. Debido a que UTF-16 requiere una o dos palabras de 16 bits por carácter, y un char solo tiene una longitud de 16 bits, algunos caracteres Unicode requieren dos chars para representarse. Esto tiene un par de consecuencias:

- La propiedad Length de una cadena puede ser mayor que el número real de caracteres.
- Un solo char no siempre es suficiente para representar completamente un carácter Unicode.

La mayoría de las aplicaciones ignoran esto porque casi todos los caracteres comúnmente utilizados se ajustan en una sección de Unicode llamada Plano Multilingüe Básico (BMP), que solo requiere una palabra de 16 bits en UTF-16. El BMP cubre varias docenas de idiomas del mundo e incluye más de 30,000 caracteres chinos. Se excluyen los caracteres de algunos lenguajes antiguos, símbolos de notación musical y algunos caracteres chinos menos comunes.

Si necesitas admitir caracteres de dos palabras, los siguientes métodos estáticos en char convierten un punto de código de 32 bits en una cadena de dos chars y viceversa:

```
string ConvertFromUtf32 (int utf32)
int    ConvertToUtf32   (char highSurrogate, char lowSurrogate)
```

Los caracteres de dos palabras se llaman sustitutos. Son fáciles de identificar porque cada palabra está en el rango de 0xD800 a 0xDFFF. Puedes utilizar los siguientes métodos estáticos en char para ayudar:

```
bool IsSurrogate     (char c)
bool IsHighSurrogate (char c)
bool IsLowSurrogate  (char c)
bool IsSurrogatePair (char highSurrogate, char lowSurrogate)
```

La clase StringInfo en el espacio de nombres System.Globalization también proporciona una variedad de métodos y propiedades para trabajar con caracteres de dos palabras.

Los caracteres fuera del BMP normalmente requieren fuentes especiales y tienen un soporte limitado por parte del sistema operativo.

Fechas y horas

Las siguientes estructuras inmutables en el espacio de nombres System representan fechas y horas: DateTime, DateTimeOffset, TimeSpan, DateOnly y TimeOnly. C# no define ninguna palabra clave especial que se ajuste a estos tipos.

TimeSpan

Un TimeSpan representa un intervalo de tiempo o una hora del día. En este último caso, es simplemente la hora del "reloj" (sin la fecha), que equivale al tiempo transcurrido desde la medianoche, sin considerar cambios de horario de verano. Un TimeSpan tiene una resolución de 100 ns, un valor máximo de aproximadamente 10 millones de días y puede ser positivo o negativo.

Hay tres formas de construir un TimeSpan:

- A través de uno de los constructores
- Llamando a uno de los métodos estáticos From...
- Restando una fecha y hora DateTime de otra

Aquí están los constructores:

```
public TimeSpan (int hours, int minutes, int seconds);
public TimeSpan (int days, int hours, int minutes, int seconds);
public TimeSpan (int days, int hours, int minutes, int seconds,
                                                   int milliseconds);
public TimeSpan (long ticks);   // Each tick = 100ns
```

Los métodos estáticos From... son más convenientes cuando quieres especificar un intervalo en una sola unidad, como minutos, horas, y así sucesivamente:

```
public static TimeSpan FromDays (double value);
public static TimeSpan FromHours (double value);
public static TimeSpan FromMinutes (double value);
public static TimeSpan FromSeconds (double value);
public static TimeSpan FromMilliseconds (double value);
```

Por ejemplo:

```
Console.WriteLine (new TimeSpan (2, 30, 0));     // 02:30:00
Console.WriteLine (TimeSpan.FromHours (2.5));    // 02:30:00
Console.WriteLine (TimeSpan.FromHours (-2.5));   // -02:30:00
```

TimeSpan sobrecarga los operadores < y >, así como los operadores + y -. La siguiente expresión evalúa a un TimeSpan de 2.5 horas:

```
TimeSpan.FromHours(2) + TimeSpan.FromMinutes(30);
```

La siguiente expresión evalúa a 1 segundo menos de 10 días:

```
TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);   // 9.23:59:59
```

Utilizando esta expresión, podemos mostrar las propiedades enteras Days, Hours, Minutes, Seconds y Milliseconds:

```
TimeSpan nearlyTenDays = TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);

Console.WriteLine (nearlyTenDays.Days);          // 9
Console.WriteLine (nearlyTenDays.Hours);         // 23
Console.WriteLine (nearlyTenDays.Minutes);       // 59
Console.WriteLine (nearlyTenDays.Seconds);       // 59
Console.WriteLine (nearlyTenDays.Milliseconds);  // 0
```

En contraste, las propiedades Total... devuelven valores de tipo double que describen el tiempo total:

```
Console.WriteLine (nearlyTenDays.TotalDays);          // 9.99998842592593
Console.WriteLine (nearlyTenDays.TotalHours);         // 239.999722222222
Console.WriteLine (nearlyTenDays.TotalMinutes);       // 14399.9833333333
Console.WriteLine (nearlyTenDays.TotalSeconds);       // 863999
Console.WriteLine (nearlyTenDays.TotalMilliseconds);  // 863999000
```

El método estático Parse hace lo contrario de ToString, convirtiendo una cadena en un TimeSpan. TryParse hace lo mismo pero devuelve false en lugar de lanzar una excepción si la conversión falla. La clase XmlConvert también proporciona métodos de conversión TimeSpan/cadena que siguen los protocolos de formato XML estándar.

El valor predeterminado para un TimeSpan es TimeSpan.Zero.

DateTime también se puede utilizar para representar la hora del día (el tiempo transcurrido desde la medianoche). Para obtener la hora actual del día, llama a DateTime.Now.TimeOfDay.

DateTime y DateTimeOffset

DateTime y DateTimeOffset son estructuras inmutables para representar una fecha y, opcionalmente, una hora. Tienen una resolución de 100 ns y abarcan los años 0001 a 9999.

DateTimeOffset es funcionalmente similar a DateTime. Su característica distintiva es que también almacena un desplazamiento de tiempo universal coordinado (UTC); esto permite resultados más significativos al comparar valores en diferentes zonas horarias.

 Hay un excelente artículo sobre la justificación de la introducción de DateTimeOffset disponible en línea, titulado "A Brief History of DateTime" por Anthony Moore.

Elección entre DateTime y DateTimeOffset

DateTime y DateTimeOffset difieren en cómo manejan las zonas horarias. Un DateTime incorpora una bandera de tres estados que indica si el DateTime es relativo a lo siguiente:

- La hora local en la computadora actual.
- UTC (equivalente moderno de la Hora Media de Greenwich).
- No especificado.

Un DateTimeOffset es más específico: almacena el desplazamiento de UTC como un TimeSpan:

```
July 01 2019 03:00:00 -06:00
```

Esto afecta las comparaciones de igualdad, que es el factor principal en la elección entre DateTime y DateTimeOffset. Específicamente:

- DateTime ignora la bandera de tres estados en las comparaciones y considera dos valores iguales si tienen el mismo año, mes, día, hora, minuto, etc.
- DateTimeOffset considera dos valores iguales si se refieren al mismo punto en el tiempo.

 El horario de verano puede hacer que esta distinción sea importante incluso si tu aplicación no necesita manejar múltiples zonas horarias geográficas.

Entonces, DateTime considera que los siguientes dos valores son diferentes, mientras que DateTimeOffset los considera iguales:

```
July 01 2019 09:00:00 +00:00 (GMT)
July 01 2019 03:00:00 -06:00 (local time, Central America)
```

En la mayoría de los casos, la lógica de igualdad de DateTimeOffset es preferible. Por ejemplo, al calcular cuál de dos eventos internacionales es más reciente, un DateTimeOffset proporciona implícitamente la respuesta correcta. De manera similar, un hacker que planea un ataque de denegación de servicio distribuido utilizaría un DateTimeOffset. Hacer lo mismo con DateTime requiere estandarizar en una única zona horaria (normalmente UTC) en toda tu aplicación.

Esto presenta dos problemas:

- Para ser amigable con el usuario final, los DateTimes en UTC requieren una conversión explícita a la hora local antes de formatear.
- Es fácil olvidarse e incorporar un DateTime local.

Sin embargo, DateTime es mejor para especificar un valor relativo a la computadora local en tiempo de ejecución. Por ejemplo, si deseas programar un archivo en cada una de tus oficinas internacionales para el próximo domingo a las 3 A.M. hora local (cuando hay menos actividad), DateTime sería más adecuado porque respetaría la hora local de cada sitio.


Internamente, DateTimeOffset utiliza un entero corto para almacenar el desplazamiento de UTC en minutos. No almacena ninguna información regional, por lo que no hay nada presente que indique si un desplazamiento de +08:00, por ejemplo, se refiere a la hora de Singapur o a la hora de Perth.

Revisitaremos las zonas horarias y las comparaciones de igualdad con más detalle en "DateOnly y TimeOnly" en la página 292.


SQL Server 2008 introdujo soporte directo para DateTimeOffset a través de un nuevo tipo de datos con el mismo nombre.

Construyendo un DateTime

DateTime define constructores que aceptan enteros para el año, mes y día, y opcionalmente la hora, minuto, segundo y milisegundo:

```
public DateTime (int year, int month, int day);

public DateTime (int year, int month, int day,
                 int hour, int minute, int second, int millisecond);
```

Si solo especificas una fecha, la hora se establece implícitamente a la medianoche (0:00).

Los constructores de DateTime también te permiten especificar un DateTimeKind, que es un enumerador con los siguientes valores:

```
Unspecified, Local, Utc
```

Esto corresponde a la bandera de tres estados descrita en la sección anterior. Unspecified es el valor predeterminado y significa que DateTime no está asociado a una zona horaria. Local significa relativo a la zona horaria local en la computadora actual. Un DateTime local no incluye información sobre a qué zona horaria específica se refiere ni, a diferencia de DateTimeOffset, el desplazamiento numérico respecto a UTC.

La propiedad Kind de un DateTime devuelve su DateTimeKind.

Los constructores de DateTime también están sobrecargados para aceptar un objeto Calendar. Esto te permite especificar una fecha utilizando cualquier subclase de Calendar definida en System.Globalization. Por ejemplo:

```
DateTime d = new DateTime (5767, 1, 1,
                           new System.Globalization.HebrewCalendar());

Console.WriteLine (d);     // 12/12/2006 12:00:00 AM
```

La presentación de la fecha en este ejemplo depende de la configuración del panel de control de tu computadora. Un DateTime siempre utiliza el calendario gregoriano por defecto. En el ejemplo anterior, se realiza una conversión única durante la construcción. Para realizar cálculos utilizando otro calendario, debes utilizar los métodos de la subclase Calendar correspondiente.

También puedes construir un DateTime con un solo valor de ticks de tipo long, donde los ticks representan la cantidad de intervalos de 100 ns desde la medianoche del 01/01/0001.

Para interoperabilidad, DateTime proporciona los métodos estáticos FromFileTime y FromFileTimeUtc para convertir desde un tiempo de archivo de Windows (especificado como un valor long) y FromOADate para convertir desde una fecha/hora de automatización OLE (especificada como un valor double).

Para construir un DateTime a partir de una cadena, puedes utilizar los métodos estáticos Parse o ParseExact. Ambos métodos aceptan indicadores opcionales y proveedores de formato. ParseExact también acepta una cadena de formato. La discusión sobre el análisis en detalle se encuentra en "Formateo y análisis" en la página 297.

Construyendo un DateTimeOffset

La construcción de un DateTimeOffset es similar. La diferencia es que también se especifica un desplazamiento de UTC como TimeSpan. Por ejemplo:

```
public DateTimeOffset (int year, int month, int day,
                       int hour, int minute, int second,
                       TimeSpan offset);

public DateTimeOffset (int year, int month, int day,
                       int hour, int minute, int second, int millisecond,
                       TimeSpan offset);
```

El TimeSpan debe ser un número entero de minutos; de lo contrario, se producirá una excepción.

DateTimeOffset también tiene constructores que aceptan un objeto Calendar, un valor de ticks de tipo long y métodos estáticos Parse y ParseExact que aceptan una cadena.

Puedes construir un DateTimeOffset a partir de un DateTime existente utilizando los siguientes constructores:

```
public DateTimeOffset (DateTime dateTime);
public DateTimeOffset (DateTime dateTime, TimeSpan offset);
```

También puedes realizar una conversión implícita utilizando el operador de asignación:

```
DateTimeOffset dt = new DateTime (2000, 2, 3);
```



La conversión implícita de DateTime a DateTimeOffset es útil porque la mayoría de las bibliotecas de .NET admiten DateTime, no DateTimeOffset.

Si no especificas un desplazamiento, se infiere del valor DateTime utilizando las siguientes reglas:

- Si el DateTime tiene un DateTimeKind de Utc, el desplazamiento es cero.
- Si el DateTime tiene un DateTimeKind de Local o Unspecified (el valor predeterminado), el desplazamiento se toma de la zona horaria local actual.

Para realizar la conversión en la dirección opuesta, DateTimeOffset proporciona tres propiedades que devuelven valores de tipo DateTime:

- La propiedad UtcDateTime devuelve un DateTime en la hora UTC.
- La propiedad LocalDateTime devuelve un DateTime en la zona horaria local actual (convirtiéndola si es necesario).
- La propiedad DateTime devuelve un DateTime en la zona horaria específica, con un Kind de Unspecified (es decir, devuelve la hora UTC más el desplazamiento).

La fecha y hora actual (The current DateTime/DateTimeOffset)

Tanto DateTime como DateTimeOffset tienen una propiedad estática Now que devuelve la fecha y hora actual:

```
Console.WriteLine (DateTime.Now);          // 11/11/2019 1:23:45 PM
Console.WriteLine (DateTimeOffset.Now);    // 11/11/2019 1:23:45 PM -06:00
```

DateTime también proporciona una propiedad Today que devuelve solo la parte de la fecha:

```
Console.WriteLine (DateTime.Today);        // 11/11/2019 12:00:00 AM
```

La propiedad estática UtcNow devuelve la fecha y hora actual en UTC:

```
Console.WriteLine (DateTime.UtcNow);        // 11/11/2019 7:23:45 AM
Console.WriteLine (DateTimeOffset.UtcNow);  // 11/11/2019 7:23:45 AM +00:00
```

La precisión de todos estos métodos depende del sistema operativo y suele estar en el rango de 10 a 20 ms.

Trabajando con fechas y horas

DateTime y DateTimeOffset proporcionan un conjunto similar de propiedades de instancia que devuelven varios elementos de fecha y hora:

```
DateTime dt = new DateTime (2000, 2, 3,
                            10, 20, 30);

Console.WriteLine (dt.Year);         // 2000
Console.WriteLine (dt.Month);        // 2
Console.WriteLine (dt.Day);          // 3
Console.WriteLine (dt.DayOfWeek);    // Thursday
Console.WriteLine (dt.DayOfYear);    // 34

Console.WriteLine (dt.Hour);         // 10
Console.WriteLine (dt.Minute);       // 20
Console.WriteLine (dt.Second);       // 30
Console.WriteLine (dt.Millisecond);  // 0
Console.WriteLine (dt.Ticks);        // 630851700300000000
Console.WriteLine (dt.TimeOfDay);    // 10:20:30  (returns a TimeSpan)
```

DateTimeOffset también tiene una propiedad Offset de tipo TimeSpan.

Ambos tipos proporcionan los siguientes métodos de instancia para realizar cálculos (la mayoría acepta un argumento de tipo double o int):

```
AddYears  AddMonths   AddDays
AddHours  AddMinutes  AddSeconds  AddMilliseconds  AddTicks
```

Todos estos métodos devuelven un nuevo DateTime o DateTimeOffset, y tienen en cuenta cosas como los años bisiestos. Puedes pasar un valor negativo para restar.

El método Add agrega un TimeSpan a un DateTime o DateTimeOffset. El operador + está sobrecargado para hacer el mismo trabajo:

```
TimeSpan ts = TimeSpan.FromMinutes (90);
Console.WriteLine (dt.Add (ts));
Console.WriteLine (dt + ts);             // same as above
```

También puedes restar un TimeSpan de un DateTime/DateTimeOffset y restar un DateTime/DateTimeOffset de otro. Este último te da un TimeSpan:

```
DateTime thisYear = new DateTime (2015, 1, 1);
DateTime nextYear = thisYear.AddYears (1);
TimeSpan oneYear = nextYear - thisYear;
```

Formato y análisis de DateTimes

Llamar a ToString en un DateTime formatea el resultado como una fecha corta (todos los números) seguida de una hora larga (incluyendo segundos). Por ejemplo:

```
11/11/2019 11:50:30 AM
```

El panel de control del sistema operativo, por defecto, determina cosas como si el día, mes o año viene primero; el uso de ceros iniciales; y si se utiliza la hora en formato de 12 o 24 horas.

Llamar a ToString en un DateTimeOffset es similar, excepto que también se devuelve la compensación de zona horaria:

```
11/11/2019 11:50:30 AM -06:00
```

Los métodos ToShortDateString y ToLongDateString devuelven solo la parte de la fecha. El formato de fecha larga también está determinado por el panel de control; un ejemplo sería "Miércoles, 11 de noviembre de 2015". Los métodos ToShortTimeString y ToLongTimeString devuelven solo la parte de la hora, como 17:10:10 (el primero excluye los segundos).

Estos cuatro métodos descritos anteriormente son en realidad accesos directos a cuatro cadenas de formato diferentes. ToString está sobrecargado para aceptar una cadena de formato y un proveedor, lo que te permite especificar una amplia gama de opciones y controlar cómo se aplican las configuraciones regionales. Esto se describe en "Formato y análisis" en la página 297.

 Los DateTimes y DateTimeOffsets pueden interpretarse incorrectamente si la configuración cultural difiere de la que estaba vigente al formatear. Puedes evitar este problema utilizando ToString en conjunto con una cadena de formato que ignore la configuración cultural (como "o"):

```
DateTime dt1 = DateTime.Now;
string cannotBeMisparsed = dt1.ToString ("o");
DateTime dt2 = DateTime.Parse (cannotBeMisparsed);
```

Los métodos estáticos Parse/TryParse y ParseExact/TryParseExact realizan lo contrario de ToString, convirtiendo una cadena en un DateTime o DateTimeOffset. Estos métodos también están sobrecargados para aceptar un proveedor de formato. Los métodos Try* devuelven false en lugar de lanzar una excepción FormatException.

Valores nulos de DateTime y DateTimeOffset

Dado que DateTime y DateTimeOffset son estructuras (struct), no son inherentemente nulos. Cuando necesitas la posibilidad de tener valores nulos, hay dos formas de solucionarlo:

- Utilizar un tipo Nullable (es decir, DateTime? o DateTimeOffset?).
- Utilizar los valores estáticos DateTime.MinValue o DateTimeOffset.MinValue (los valores predeterminados para estos tipos).

Por lo general, el uso de un tipo Nullable es el enfoque más recomendado, ya que el compilador ayuda a prevenir errores. DateTime.MinValue es útil para la compatibilidad con versiones anteriores de código escrito antes de C# 2.0 (cuando se introdujeron los tipos de valor nulos).

 Llamar a ToUniversalTime o ToLocalTime en DateTime.MinValue puede hacer que ya no sea DateTime.MinValue (dependiendo de qué lado del GMT te encuentres). Si estás justo en GMT (Inglaterra, fuera del horario de verano), el problema no surgirá en absoluto porque las horas locales y UTC son las mismas. ¡Esta es tu compensación por el invierno en Inglaterra!

DateOnly y TimeOnly

En .NET 6, existen las estructuras DateOnly y TimeOnly para representar exclusivamente una fecha o una hora.

DateOnly es similar a DateTime, pero sin componente de hora. DateOnly también carece de DateTimeKind; en efecto, siempre se considera como No especificado (Unspecified) y no tiene concepto de Local o Utc. La alternativa histórica a DateOnly era utilizar DateTime con una hora cero (medianoche). La dificultad con este enfoque es que las comparaciones de igualdad fallan cuando una hora distinta de cero se introduce en el código.

TimeOnly es similar a DateTime, pero sin componente de fecha. TimeOnly está diseñado para capturar la hora del día y es adecuado para aplicaciones como el registro de horas de alarma o horarios de apertura.

DateTime y las zonas horarias

DateTime es bastante simplista en su manejo de las zonas horarias. Internamente, almacena un DateTime utilizando dos piezas de información:

- Un número de 62 bits que indica el número de ticks desde el 01/01/0001.
- Un enum de 2 bits que indica el DateTimeKind (No especificado, Local o Utc).

Cuando comparas dos instancias de DateTime, solo se comparan sus valores de ticks; se ignoran sus DateTimeKinds:

```
DateTime dt1 = new DateTime (2000, 1, 1, 10, 20, 30, DateTimeKind.Local);
DateTime dt2 = new DateTime (2000, 1, 1, 10, 20, 30, DateTimeKind.Utc);
Console.WriteLine (dt1 == dt2);          // True
DateTime local = DateTime.Now;
DateTime utc = local.ToUniversalTime();
Console.WriteLine (local == utc);        // False
```

Los métodos de instancia ToUniversalTime/ToLocalTime convierten a hora universal/local. Estos métodos aplican la configuración de zona horaria actual de la computadora y devuelven un nuevo DateTime con DateTimeKind Utc o Local. No ocurre ninguna conversión si llamas a ToUniversalTime en un DateTime que ya es Utc, o a ToLocalTime en un DateTime que ya es Local. Sin embargo, se realizará una conversión si llamas a ToUniversalTime o ToLocalTime en un DateTime que es No especificado (Unspecified).

Puedes construir un DateTime que difiera en Kind de otro utilizando el método estático DateTime.SpecifyKind:

```
DateTime d = new DateTime (2015, 12, 12);  // Unspecified
DateTime utc = DateTime.SpecifyKind (d, DateTimeKind.Utc);
Console.WriteLine (utc);            // 12/12/2015 12:00:00 AM
```

DateTimeOffset y las zonas horarias

Internamente, DateTimeOffset consta de un campo DateTime cuyo valor siempre está en UTC, y un campo entero de 16 bits para el desplazamiento UTC en minutos. Las comparaciones se realizan solo en el DateTime (UTC); el desplazamiento se utiliza principalmente para el formateo.

Los métodos ToUniversalTime/ToLocalTime devuelven un DateTimeOffset que representa el mismo punto en el tiempo pero con un desplazamiento UTC o local. A diferencia de DateTime, estos métodos no afectan el valor subyacente de fecha/hora, solo el desplazamiento:

```
DateTimeOffset local = DateTimeOffset.Now;
DateTimeOffset utc   = local.ToUniversalTime();

Console.WriteLine (local.Offset);   // -06:00:00 (in Central America)
Console.WriteLine (utc.Offset);     // 00:00:00

Console.WriteLine (local == utc);                // True
```

Para incluir el desplazamiento en la comparación, debes utilizar el método EqualsExact:

```
Console.WriteLine (local.EqualsExact (utc));      // False
```

TimeZoneInfo

La clase TimeZoneInfo proporciona información sobre nombres de zonas horarias, desplazamientos UTC y reglas de horario de verano.

TimeZone

El método estático TimeZone.CurrentTimeZone devuelve una TimeZone:

```
TimeZone zone = TimeZone.CurrentTimeZone;
Console.WriteLine (zone.StandardName);      // Pacific Standard Time
Console.WriteLine (zone.DaylightName);      // Pacific Daylight Time
```

El método GetDaylightChanges devuelve información específica de horario de verano para un año determinado:

```
DaylightTime day = zone.GetDaylightChanges (2019);
Console.WriteLine (day.Start.ToString ("M"));       // 10 March
Console.WriteLine (day.End.ToString ("M"));         // 03 November
Console.WriteLine (day.Delta);                      // 01:00:00
```

TimeZoneInfo

El método estático TimeZoneInfo.Local devuelve un objeto TimeZoneInfo basado en la configuración local actual. Lo siguiente demuestra el resultado si se ejecuta en California:

```
TimeZoneInfo zone = TimeZoneInfo.Local;
Console.WriteLine (zone.StandardName);      // Pacific Standard Time
Console.WriteLine (zone.DaylightName);      // Pacific Daylight Time
```

Los métodos IsDaylightSavingTime y GetUtcOffset funcionan de la siguiente manera:

```
  DateTime dt1 = new DateTime (2019, 1, 1);   // DateTimeOffset works, too
  DateTime dt2 = new DateTime (2019, 6, 1);
  Console.WriteLine (zone.IsDaylightSavingTime (dt1));     // True
  Console.WriteLine (zone.IsDaylightSavingTime (dt2));     // False
  Console.WriteLine (zone.GetUtcOffset (dt1));             // -08:00:00
  Console.WriteLine (zone.GetUtcOffset (dt2));             // -07:00:00
```

Puedes obtener un TimeZoneInfo para cualquier zona horaria del mundo llamando a FindSystemTimeZoneById con el ID de la zona. Cambiaremos a Australia Occidental por razones que pronto se aclararán:

```
TimeZoneInfo wa = TimeZoneInfo.FindSystemTimeZoneById
                  ("W. Australia Standard Time");

Console.WriteLine (wa.Id);                  // W. Australia Standard Time
Console.WriteLine (wa.DisplayName);         // (GMT+08:00) Perth
Console.WriteLine (wa.BaseUtcOffset);       // 08:00:00
Console.WriteLine (wa.SupportsDaylightSavingTime);     // True
```

La propiedad Id corresponde al valor pasado a FindSystemTimeZoneById. El método estático GetSystemTimeZones devuelve todas las zonas horarias del mundo; por lo tanto, puedes listar todas las cadenas de ID de zona válidas de la siguiente manera:

```
foreach (TimeZoneInfo z in TimeZoneInfo.GetSystemTimeZones())
  Console.WriteLine (z.Id);
```



También puedes crear una zona horaria personalizada llamando a TimeZoneInfo.CreateCustomTimeZone. Dado que TimeZoneInfo es inmutable, debes pasar todos los datos relevantes como argumentos del método.

Puedes serializar una zona horaria predefinida o personalizada a una cadena (semi) legible por humanos llamando a ToSerializedString, y deserializarla llamando a TimeZoneInfo.FromSerializedString.

El método estático ConvertTime convierte un DateTime o DateTimeOffset de una zona horaria a otra. Puedes incluir solo una TimeZoneInfo de destino, o tanto la TimeZoneInfo de origen como la de destino. También puedes convertir directamente desde o hacia UTC con los métodos ConvertTimeFromUtc y ConvertTimeToUtc.

Para trabajar con el horario de verano, TimeZoneInfo proporciona los siguientes métodos adicionales:

- IsInvalidTime devuelve true si un DateTime está dentro de la hora (o delta) que se salta cuando los relojes avanzan.

- IsAmbiguousTime devuelve true si un DateTime o DateTimeOffset está dentro de la hora (o delta) que se repite cuando los relojes retroceden.
- GetAmbiguousTimeOffsets devuelve una matriz de TimeSpans que representa las opciones de desplazamiento válidas para un DateTime o DateTimeOffset ambiguo.

No se pueden obtener fechas simples de un TimeZoneInfo que indiquen el inicio y fin del Horario de Verano. En su lugar, debes llamar a GetAdjustmentRules, que devuelve un resumen declarativo de todas las reglas de horario de verano que se aplican a todos los años. Cada regla tiene una DateStart y una DateEnd que indican el rango de fechas en el cual la regla es válida:

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules())
  Console.WriteLine ("Rule: applies from " + rule.DateStart +
                                     " to " + rule.DateEnd);
```

Western Australia implementó por primera vez el Horario de Verano en 2006, en medio de la temporada (y luego lo canceló en 2009). Esto requirió una regla especial para el primer año, por lo tanto, hay dos reglas:

```
Rule: applies from 1/01/2006 12:00:00 AM to 31/12/2006 12:00:00 AM
Rule: applies from 1/01/2007 12:00:00 AM to 31/12/2009 12:00:00 AM
```

Cada AdjustmentRule tiene una propiedad DaylightDelta de tipo TimeSpan (generalmente es de una hora) y propiedades llamadas DaylightTransitionStart y DaylightTransitionEnd. Estas dos últimas son del tipo TimeZoneInfo.TransitionTime, que tiene las siguientes propiedades:

```
public bool IsFixedDateRule { get; }
public DayOfWeek DayOfWeek { get; }
public int Week { get; }
public int Day { get; }
public int Month { get; }
public DateTime TimeOfDay { get; }
```

Una hora de transición es algo complicada ya que necesita representar tanto fechas fijas como flotantes. Un ejemplo de una fecha flotante sería "el último domingo de marzo". Aquí están las reglas para interpretar una hora de transición:

1. Si, para una transición de fin, IsFixedDateRule es verdadero, Day es 1, Month es 1 y TimeOfDay es DateTime.MinValue, eso significa que no hay fin del Horario de Verano en ese año (esto solo puede suceder en el hemisferio sur, al momento de la implementación inicial del horario de verano en una región).

2. De lo contrario, si IsFixedDateRule es verdadero, las propiedades Month, Day y TimeOfDay determinan el inicio o fin de la regla de ajuste.

3. De lo contrario, si IsFixedDateRule es falso, las propiedades Month, DayOfWeek, Week y TimeOfDay determinan el inicio o fin de la regla de ajuste.

En el último caso, Week se refiere a la semana del mes, siendo "5" el último semana. Podemos demostrar esto enumerando las reglas de ajuste para nuestra zona horaria de Western Australia (wa):

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules())
{
  Console.WriteLine ("Rule: applies from " + rule.DateStart +
                                     " to " + rule.DateEnd);

  Console.WriteLine ("   Delta: " + rule.DaylightDelta);

   Console.WriteLine ("   Start: " + FormatTransitionTime
                                      (rule.DaylightTransitionStart, false));

   Console.WriteLine ("   End:   " + FormatTransitionTime
                                      (rule.DaylightTransitionEnd, true));
   Console.WriteLine();
 }
```

En el método FormatTransitionTime, seguimos las reglas que acabamos de describir:

```
static string FormatTransitionTime (TimeZoneInfo.TransitionTime tt,
                                    bool endTime)
{
  if (endTime && tt.IsFixedDateRule
              && tt.Day == 1 && tt.Month == 1
              && tt.TimeOfDay == DateTime.MinValue)
    return "-";

  string s;
  if (tt.IsFixedDateRule)
    s = tt.Day.ToString();
  else
    s = "The " +
        "first second third fourth last".Split() [tt.Week - 1] +
        " " + tt.DayOfWeek + " in";

  return s + " " + DateTimeFormatInfo.CurrentInfo.MonthNames [tt.Month-1]
           + " at " + tt.TimeOfDay.TimeOfDay;
}
```

Horario de Verano y DateTime

Si utilizas DateTimeOffset o DateTime en UTC, las comparaciones de igualdad no se ven afectadas por el Horario de Verano. Sin embargo, con DateTime local, el horario de verano puede ser problemático.

Podemos resumir las reglas de la siguiente manera:

- El horario de verano afecta la hora local pero no la hora UTC.
- Cuando los relojes retroceden, las comparaciones que dependen de que el tiempo avance se romperán si (y solo si) utilizan DateTime local.
- Siempre puedes realizar conversiones confiables entre UTC y la hora local (en la misma computadora), incluso cuando los relojes retroceden.

El método IsDaylightSavingTime te indica si un DateTime local dado está sujeto al Horario de Verano. Los tiempos en UTC siempre devolverán false:

```
Console.Write (DateTime.Now.IsDaylightSavingTime());     // True or False
Console.Write (DateTime.UtcNow.IsDaylightSavingTime());  // Always False
```

Suponiendo que dto es un DateTimeOffset, la siguiente expresión hace lo mismo:

```
dto.LocalDateTime.IsDaylightSavingTime
```

El final del Horario de Verano presenta una complicación particular para los algoritmos que utilizan la hora local, porque cuando los relojes retroceden, la misma hora (o más precisamente, el Delta) se repite.



Puedes comparar de manera confiable cualquier par de DateTimes llamando primero a ToUniversalTime en cada uno. Esta estrategia falla si (y solo si) exactamente uno de ellos tiene un DateTimeKind de Unspecified. Esta posibilidad de fallo es otra razón para preferir DateTimeOffset.

Formateo y Análisis

El formateo implica convertir a una cadena; el análisis implica convertir desde una cadena. La necesidad de formatear o analizar surge con frecuencia en la programación, en una variedad de situaciones. Por lo tanto, .NET proporciona una variedad de mecanismos:

ToString y Parse

Estos métodos proporcionan funcionalidad predeterminada para muchos tipos.

Proveedores de formato

Estos se manifiestan como métodos adicionales ToString (y Parse) que aceptan una cadena de formato y/o un proveedor de formato. Los proveedores de formato son altamente flexibles y conscientes de la cultura. .NET incluye proveedores de formato para los tipos numéricos y DateTime/DateTimeOffset.

XmlConvert

Esta es una clase estática con métodos que formatean y analizan respetando los estándares XML. XmlConvert también es útil para conversiones de propósito general cuando se necesita independencia de la cultura o se desea prevenir errores de análisis. XmlConvert admite los tipos numéricos, bool, DateTime, DateTimeOffset, TimeSpan y Guid.

Convertidores de tipo

Estos están dirigidos a diseñadores y analizadores XAML.

En esta sección, discutiremos los dos primeros mecanismos, centrándonos particularmente en los proveedores de formato. Luego describiremos XmlConvert, los convertidores de tipo y otros mecanismos de conversión.

ToString y Parse

El mecanismo de formateo más simple es el método ToString. Proporciona una salida significativa en todos los tipos de valores simples (bool, DateTime, DateTimeOffset, TimeSpan, Guid y todos los tipos numéricos). Para la operación inversa, cada uno de estos tipos define un método Parse estático:

```
string s = true.ToString();     // s = "True"
bool b = bool.Parse (s);        // b = true
```

Si el análisis falla, se lanzará una excepción FormatException. Muchos tipos también definen un método TryParse, que devuelve false si la conversión falla en lugar de lanzar una excepción:

```
bool failure = int.TryParse ("qwerty", out int i1);
bool success = int.TryParse ("123", out int i2);
```

Si no te importa el resultado y solo quieres probar si el análisis tendría éxito, puedes usar un descarte:

```
bool success = int.TryParse ("123", out int _);
```

Si anticipas un error, llamar a TryParse es más rápido y elegante que llamar a Parse dentro de un bloque de manejo de excepciones.

Los métodos Parse y TryParse en DateTime(Offset) y los tipos numéricos respetan la configuración cultural local; puedes cambiar esto especificando un objeto CultureInfo. Es aconsejable especificar la cultura invariante. Por ejemplo, el análisis de "1.234" en un double nos da 1234 en Alemania:

```
Console.WriteLine (double.Parse ("1.234"));   // 1234  (In Germany)
```

Esto se debe a que en Alemania, el punto indica un separador de miles en lugar de un punto decimal. Especificar la cultura invariante soluciona esto:

```
double x = double.Parse ("1.234", CultureInfo.InvariantCulture);
```

Lo mismo se aplica al llamar a ToString():

```
string x = 1.234.ToString (CultureInfo.InvariantCulture);
```

Proveedores de formato

A veces, necesitas tener un mayor control sobre cómo se realiza el formateo y el análisis. Hay docenas de formas de formatear un DateTime(Offset), por ejemplo. Los proveedores de formato permiten un control extenso sobre el formateo y el análisis, y son compatibles con los tipos numéricos y las fechas/horas. Los proveedores de formato también son utilizados por los controles de interfaz de usuario para el formateo y análisis.

La puerta de entrada para utilizar un proveedor de formato es IFormattable. Todos los tipos numéricos y DateTime(Offset) implementan esta interfaz:

```
public interface IFormattable
{
  string ToString (string format, IFormatProvider formatProvider);
}
```

El primer argumento es la cadena de formato y el segundo es el proveedor de formato. La cadena de formato proporciona instrucciones y el proveedor de formato determina cómo se traducen las instrucciones. Por ejemplo:

```
NumberFormatInfo f = new NumberFormatInfo();
f.CurrencySymbol = "$$";
Console.WriteLine (3.ToString ("C", f));          // $$ 3.00
```

Aquí, "C" es una cadena de formato que indica moneda, y el objeto NumberFormatInfo es un proveedor de formato que determina cómo se representan la moneda y otras representaciones numéricas. Este mecanismo permite la globalización.



Todas las cadenas de formato para números y fechas se enumeran en "Standard Format Strings and Parsing Flags" en la página 303.

Si especificas una cadena de formato o un proveedor nulo, se aplica un valor predeterminado. El proveedor de formato predeterminado es CultureInfo.CurrentCulture, que, a menos que se reasigne, refleja la configuración del panel de control en tiempo de ejecución de la computadora. Por ejemplo, en esta computadora:

```
Console.WriteLine (10.3.ToString ("C", null));  // $10.30
```

Para mayor comodidad, la mayoría de los tipos sobrecargan ToString de manera que puedes omitir un proveedor nulo:

```
Console.WriteLine (10.3.ToString ("C"));     // $10.30
Console.WriteLine (10.3.ToString ("F4"));    // 10.3000 (Fix to 4 D.P.)
```

Llamar a ToString en un DateTime(Offset) o un tipo numérico sin argumentos es equivalente a usar un proveedor de formato predeterminado con una cadena de formato vacía.

.NET define tres proveedores de formato (todos ellos implementan IFormatProvider):

```
NumberFormatInfo
DateTimeFormatInfo
CultureInfo
```



Todos los tipos enumerados también son formateables, aunque no hay una clase IFormatProvider especial.

Proveedores de formato y CultureInfo

Dentro del contexto de los proveedores de formato, CultureInfo actúa como un mecanismo de indirección para los otros dos proveedores de formato, devolviendo un objeto NumberFormatInfo o DateTimeFormatInfo aplicable a la configuración regional de la cultura.

En el siguiente ejemplo, solicitamos una cultura específica (idioma inglés en Gran Bretaña):

```
CultureInfo uk = CultureInfo.GetCultureInfo ("en-GB");
Console.WriteLine (3.ToString ("C", uk));      // £3.00
```

Esto se ejecuta utilizando el objeto NumberFormatInfo predeterminado aplicable a la cultura en-GB.

El siguiente ejemplo formatea un DateTime con la cultura invariante. La cultura invariante siempre es la misma, independientemente de la configuración de la computadora:

```
DateTime dt = new DateTime (2000, 1, 2);
CultureInfo iv = CultureInfo.InvariantCulture;
Console.WriteLine (dt.ToString (iv));            // 01/02/2000 00:00:00
Console.WriteLine (dt.ToString ("d", iv));       // 01/02/2000
```



La cultura invariante se basa en la cultura estadounidense, con las siguientes diferencias:

- El símbolo de la moneda es ¤ en lugar de $.
- Las fechas y horas se formatean con ceros a la izquierda (aunque todavía se muestra el mes primero).
- La hora utiliza el formato de 24 horas en lugar de un indicador AM/PM.

Usando NumberFormatInfo o DateTimeFormatInfo

En el siguiente ejemplo, creamos una instancia de NumberFormatInfo y cambiamos el separador de grupos de una coma a un espacio. Luego lo utilizamos para formatear un número con tres decimales:

```
NumberFormatInfo f = new NumberFormatInfo ();
f.NumberGroupSeparator = " ";
Console.WriteLine (12345.6789.ToString ("N3", f));   // 12 345.679
```

La configuración inicial de un NumberFormatInfo o DateTimeFormatInfo se basa en la cultura invariante. Sin embargo, a veces es más útil elegir un punto de partida diferente. Para hacer esto, puedes clonar un proveedor de formato existente:

```
NumberFormatInfo f = (NumberFormatInfo)
                     CultureInfo.CurrentCulture.NumberFormat.Clone();
```

Un proveedor de formato clonado siempre es editable, incluso si el original era de solo lectura.

Formato compuesto

Las cadenas de formato compuesto te permiten combinar la sustitución de variables con cadenas de formato. El método estático string.Format acepta una cadena de formato compuesto (lo ilustramos en "String.Format y cadenas de formato compuesto" en la página 276):

```
string composite = "Credit={0:C}";
Console.WriteLine (string.Format (composite, 500));   // Credit=$500.00
```

La clase Console sobrecarga sus métodos Write y WriteLine para aceptar cadenas de formato compuesto, lo que nos permite acortar un poco este ejemplo:

```
Console.WriteLine ("Credit={0:C}", 500);   // Credit=$500.00
```

También puedes agregar una cadena de formato compuesto a un StringBuilder (mediante AppendFormat) y a un TextWriter para E/S (consulta el Capítulo 15).

string.Format acepta un proveedor de formato opcional. Una aplicación sencilla de esto es llamar a ToString en un objeto arbitrario mientras se pasa un proveedor de formato:

```
string s = string.Format (CultureInfo.InvariantCulture, "{0}", someObject);
```

Esto es equivalente a lo siguiente:

```
string s;
if (someObject is IFormattable)
  s = ((IFormattable)someObject).ToString (null,
                                           CultureInfo.InvariantCulture);
else if (someObject == null)
  s = "";
else
  s = someObject.ToString();
```

Análisis con proveedores de formato

No existe una interfaz estándar para el análisis a través de un proveedor de formato. En su lugar, cada tipo participante sobrecarga su método estático Parse (y TryParse) para aceptar un proveedor de formato y, opcionalmente, un enum NumberStyles o DateTimeStyles.

NumberStyles y DateTimeStyles controlan cómo funciona el análisis, permitiéndote especificar cosas como si se pueden incluir paréntesis o símbolos de moneda en la cadena de entrada (por defecto, la respuesta a ambas preguntas es no). Por ejemplo:

```
int error = int.Parse ("(2)");   // Exception thrown

int minusTwo = int.Parse ("(2)", NumberStyles.Integer |
                                 NumberStyles.AllowParentheses);   // OK

decimal fivePointTwo = decimal.Parse ("£5.20", NumberStyles.Currency,
                       CultureInfo.GetCultureInfo ("en-GB"));
```

La siguiente sección enumera todos los miembros de NumberStyles y DateTimeStyles, así como las reglas de análisis predeterminadas para cada tipo.

IFormatProvider e ICustomFormatter

Todos los proveedores de formato implementan IFormatProvider:

```
public interface IFormatProvider { object GetFormat (Type formatType); }
```

El propósito de este método es proporcionar indirección, lo que permite que CultureInfo delegue en un objeto NumberFormatInfo o DateTimeFormatInfo adecuado para realizar el trabajo.

Al implementar IFormatProvider, junto con ICustomFormatter, también puedes escribir tu propio proveedor de formato que funcione en conjunto con los tipos existentes.

ICustomFormatter define un método único de la siguiente manera:

```
string Format (string format, object arg, IFormatProvider formatProvider);
```

El siguiente proveedor de formato personalizado escribe los números en palabras:

```
public class WordyFormatProvider : IFormatProvider, ICustomFormatter
{
  static readonly string[] _numberWords =

    "zero one two three four five six seven eight nine minus point".Split();

  IFormatProvider _parent;   // Allows consumers to chain format providers

  public WordyFormatProvider () : this (CultureInfo.CurrentCulture) { }
  public WordyFormatProvider (IFormatProvider parent) => _parent = parent;

  public object GetFormat (Type formatType)
  {
    if (formatType == typeof (ICustomFormatter)) return this;
    return null;
  }

  public string Format (string format, object arg, IFormatProvider prov)
  {
    // If it's not our format string, defer to the parent provider:
    if (arg == null || format != "W")
      return string.Format (_parent, "{0:" + format + "}", arg);

    StringBuilder result = new StringBuilder();
    string digitList = string.Format (CultureInfo.InvariantCulture,
                                      "{0}", arg);
    foreach (char digit in digitList)
    {
      int i = "0123456789-.".IndexOf (digit),
                                      StringComparison.InvariantCulture);
      if (i == -1) continue;
      if (result.Length > 0) result.Append (' ');
      result.Append (_numberWords[i]);
    }
    return result.ToString();
  }
}
```

Observa que en el método Format, usamos string.Format, con InvariantCulture, para convertir el número de entrada a una cadena. Hubiera sido más sencillo llamar simplemente a ToString() en arg, pero en ese caso se habría utilizado CurrentCulture en su lugar. La razón por la que necesitamos la cultura invariante es evidente unas líneas más adelante:

```
int i = "0123456789-.".IndexOf (digit),
                                        StringComparison.InvariantCulture);
```

Aquí es crítico que la cadena de números solo contenga los caracteres 0123456789- y no sus versiones internacionalizadas.

Aquí tienes un ejemplo de uso de WordyFormatProvider:

```
double n = -123.45;
IFormatProvider fp = new WordyFormatProvider();
Console.WriteLine (string.Format (fp, "{0:C} in words is {0:W}", n));

// -$123.45 in words is minus one two three point four five
```

Solo puedes utilizar proveedores de formato personalizados en cadenas de formato compuestas.

Las cadenas de formato estándar y los indicadores de análisis

Las cadenas de formato estándar controlan cómo se convierte un tipo numérico o DateTime/DateTimeOffset en una cadena. Hay dos tipos de cadenas de formato:

Cadenas de formato estándar

Con estas, proporcionas una guía general. Una cadena de formato estándar consiste en una única letra, seguida opcionalmente de un dígito (cuyo significado depende de la letra). Un ejemplo es "C" o "F2".

Cadenas de formato personalizadas

Con estas, puedes personalizar cada carácter con una plantilla. Un ejemplo es

`"0:#.000E+00".`

Las cadenas de formato personalizadas no están relacionadas con los proveedores de formato personalizados.

Cadenas de formato numérico

La Tabla 6-2 enumera todas las cadenas de formato numérico estándar.

Table 6-2. Standard numeric format strings

Letter	Meaning	Sample input	Result	Notes

G or g	"General"	`1.2345, "G"`
		`0.00001, "G"`
		`0.00001, "g"`
		`1.2345, "G3"`
		`12345, "G3"`
			`1.2345`
			`1E-05`
			`1e-05`
			`1.23`
			`1.23E04`
				Switches to exponential notation for small or large numbers.
				G3 limits precision to three digits in *total* (before + after point).

F	Fixed point	`2345.678, "F2"` `2345.6, "F2"`	`2345.68` `2345.60`	F2 rounds to two decimal places.
N	Fixed point with *group separator* ("Numeric")	`2345.678, "N2"` `2345.6, "N2"`	`2,345.68` `2,345.60`	As above, with group (1,000s) separator (details from format provider).
D	Pad with leading zeros	`123, "D5"` `123, "D1"`	`00123` `123`	For integral types only. D5 pads left to five digits; does not truncate.

E or e	Force exponential notation	`56789, "E"`
		`56789, "e"`
		`56789, "E2"`
			`5.678900E+004`
			`5.678900e+004`
			`5.68E+004`
				Six-digit default precision.

C	Currency	`1.2, "C"` `1.2, "C4"`	`$1.20` `$1.2000`	C with no digit uses default number of D.P. from format provider.
P	Percent	`.503, "P"` `.503, "P0"`	`50.30%` `50%`	Uses symbol and layout from format provider. Decimal places can optionally be overridden.

X or x	Hexadecimal	`47, "X"`
		`47, "x"`
		`47, "X4"`
			`2F`
			`2f`
			`002F`
				X for uppercase hex digits; x for lowercase hex digits.
				Integrals only.

R or G9/G17	Round-trip	`1f / 3f, "R"`	`0.333333343`	Use R for `BigInteger`, G17 for `double`, or G9 for `float`.

El suministro de una cadena de formato numérico nula, en blanco o sin formato es equivalente al uso de la cadena de formato estándar "G" seguida de ningún dígito. Esto muestra el siguiente comportamiento:

- Los números más pequeños que 10^-4 o más grandes que la precisión del tipo se expresan en notación exponencial (científica).
- Los dos decimales en el límite de precisión de float o double se redondean para ocultar las imprecisiones inherentes en la conversión a decimal desde su forma binaria subyacente.



El redondeo automático descrito anteriormente suele ser beneficioso y pasa desapercibido. Sin embargo, puede causar problemas si necesitas preservar la igualdad de valor al convertir un número a una cadena y luego nuevamente a un número (tal vez de manera repetida). Por esta razón, existen las cadenas de formato R, G17 y G9 para evitar este redondeo implícito.

La Tabla 6-3 enumera las cadenas de formato numérico personalizadas.

Table 6-3. Custom numeric format strings

Specifier	Meaning	Sample input	Result	Notes
#	Digit placeholder	`12.345, ".##"` `12.345, ".####"`	`12.35` `12.345`	Limits digits after D.P.

0	Zero placeholder	`12.345, ".00"`
		`12.345, ".0000"`
		`99, "000.00"`
			`12.35`
			`12.3450`
			`099.00`
				As above, but also pads with zeros before and after D.P.

.	Decimal point			Indicates D.P. Actual symbol comes from `NumberFormatInfo`.
,	Group separator	`1234, "#,###,###"` `1234, "0,000,000"`	`1,234` `0,001,234`	Symbol comes from `NumberFormatInfo`.
, (as above)	Multiplier	`1000000, "#,"` `1000000, "#,,`	`1000` `1`	If comma is at end or before D.P., it acts as a multiplier—dividing result by 1,000, 1,000,000, etc.

Specifier	Meaning	Sample input	Result	Notes
%	Percent notation	0.6, "00%"	60%	First multiplies by 100 and then substitutes percent symbol obtained from NumberFormatInfo.
E0, e0, E+0, e+0 E-0, e-0	Exponent notation	1234, "0E0" 1234, "0E+0" 1234, "0.00E00" 1234, "0.00e00"	1E3 1E+3 1.23E03 1.23e03	
\	Literal character quote	50, @"\#0"	#50	Use in conjunction with an @ prefix on the string—or use \\.
'xx''xx'	Literal string quote	50, "0 '...'"	50 ...	
;	Section separator	15, "#;(#);zero"	15	(If positive.)
		-5, "#;(#);zero"	(5)	(If negative.)
		0, "#;(#);zero"	zero	(If zero.)
Any other char	Literal	35.2, "$0 . 00c"	$35 . 20c	

NumberStyles

Cada tipo numérico define un método Parse estático que acepta un argumento NumberStyles. NumberStyles es un enum de banderas que te permite determinar cómo se lee la cadena al convertirla a un tipo numérico. Tiene los siguientes miembros combinables:

```
AllowLeadingWhite    AllowTrailingWhite
AllowLeadingSign     AllowTrailingSign
AllowParentheses     AllowDecimalPoint
AllowThousands       AllowExponent
AllowCurrencySymbol  AllowHexSpecifier
```

NumberStyles también define los siguientes miembros compuestos:

```
None  Integer  Float  Number  HexNumber  Currency  Any
```

Excepto por None, todos los valores compuestos incluyen AllowLeadingWhite y AllowTrailingWhite. La Figura 6-1 muestra su composición restante, destacando los tres más útiles.

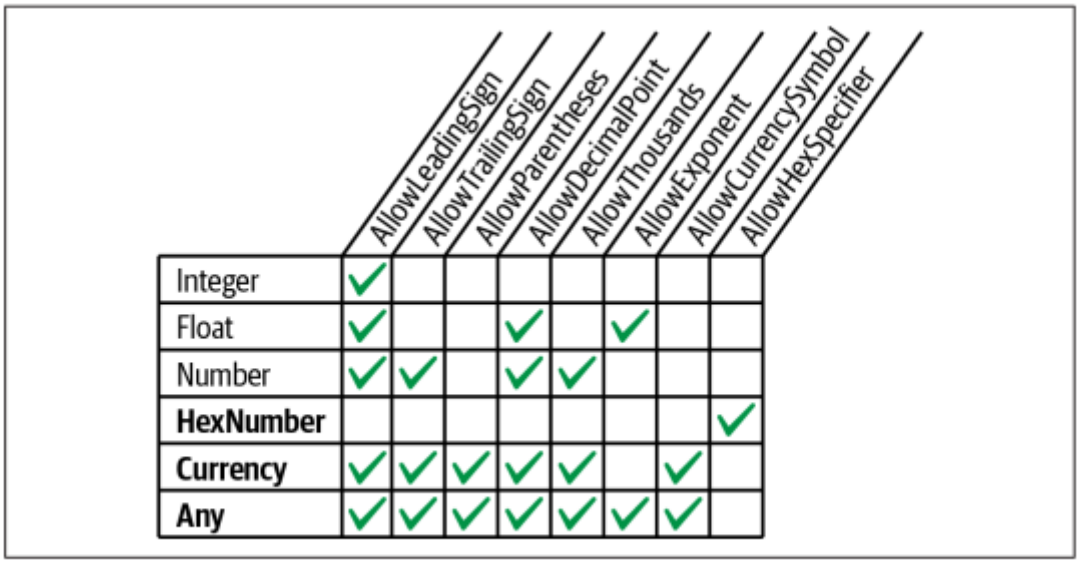


Figure 6-1. Composite NumberStyles

Cuando llamas a Parse sin especificar ninguna bandera, se aplican los valores predeterminados ilustrados en la Figura 6-2.

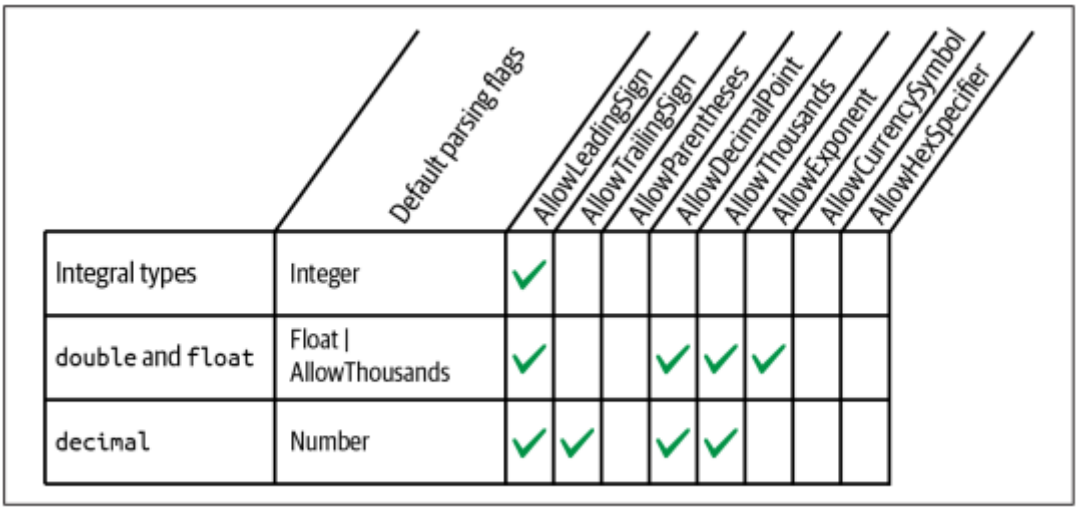


Figure 6-2. Default parsing flags for numeric types

Si no deseas utilizar los valores predeterminados mostrados en la Figura 6-2, debes especificar explícitamente NumberStyles. Por ejemplo:

```
int thousand = int.Parse ("3E8", NumberStyles.HexNumber);
int minusTwo = int.Parse ("(2)", NumberStyles.Integer |
                                 NumberStyles.AllowParentheses);
double aMillion = double.Parse ("1,000,000", NumberStyles.Any);
decimal threeMillion = decimal.Parse ("3e6", NumberStyles.Any);
decimal fivePointTwo = decimal.Parse ("$5.20", NumberStyles.Currency);
```

Dado que no hemos especificado un proveedor de formato, este ejemplo funciona con el símbolo de moneda, el separador de grupo, el punto decimal, etc., locales. El siguiente ejemplo está codificado para funcionar con el símbolo del euro y un separador de grupo en blanco para las monedas:

```
NumberFormatInfo ni = new NumberFormatInfo();
ni.CurrencySymbol = "€";
ni.CurrencyGroupSeparator = " ";
double million = double.Parse ("€1 000 000", NumberStyles.Currency, ni);
```

Formato de fecha/hora

Las cadenas de formato para DateTime/DateTimeOffset se pueden dividir en dos grupos según si respetan la configuración de la cultura y el proveedor de formato. La Tabla 6-4 enumera aquellos que lo hacen; la Tabla 6-5 enumera aquellos que no lo hacen. La salida de muestra proviene de dar formato al siguiente DateTime (con cultura invariante, en el caso de la Tabla 6-4):

```
new DateTime (2000, 1, 2,  17, 18, 19);
```

Table 6-4. Culture-sensitive date/time format strings

Format string	Meaning	Sample output
d	Short date	01/02/2000
D	Long date	Sunday, 02 January 2000
t	Short time	17:18
T	Long time	17:18:19
f	Long date + short time	Sunday, 02 January 2000 17:18
F	Long date + long time	Sunday, 02 January 2000 17:18:19
g	Short date + short time	01/02/2000 17:18
G (default)	Short date + long time	01/02/2000 17:18:19
m, M	Month and day	02 January
y, Y	Year and month	January 2000

Table 6-5. Culture-insensitive date/time format strings

Format string	Meaning	Sample output	Notes
o	Round-trippable	2000-01-02T17:18:19.0000000	Will append time zone information unless DateTimeKind is Unspecified.
r, R	RFC 1123 standard	Sun, 02 Jan 2000 17:18:19 GMT	You must explicitly convert to UTC with DateTime.ToUniversalTime.
s	Sortable; ISO 8601	2000-01-02T17:18:19	Compatible with text-based sorting.
u	"Universal" sortable	2000-01-02 17:18:19Z	Similar to above; must explicitly convert to UTC.
U	UTC	Sunday, 02 January 2000 17:18:19	Long date + short time, converted to UTC.

Las cadenas de formato "r", "R" y "u" emiten un sufijo que implica UTC; sin embargo, no convierten automáticamente una fecha y hora local a UTC (por lo que debes realizar la conversión tú mismo). Irónicamente, "U" convierte automáticamente a UTC pero no escribe un sufijo de zona horaria. De hecho, "o" es el único especificador de formato en el grupo que puede escribir un DateTime sin ambigüedades sin intervención.

DateTimeFormatInfo también admite cadenas de formato personalizadas, que son análogas a las cadenas de formato personalizadas para números. La lista es extensa y está disponible en línea en la documentación de Microsoft. Aquí tienes un ejemplo de una cadena de formato personalizada:

```
yyyy-MM-dd HH:mm:ss
```

Análisis incorrecto de DateTimes

Las cadenas que colocan el mes o el día primero son ambiguas y pueden interpretarse incorrectamente fácilmente, especialmente si tienes clientes globales. Esto no es un problema en los controles de interfaz de usuario, porque se aplican las mismas configuraciones al analizar que al dar formato. Sin embargo, al escribir en un archivo, por ejemplo, el análisis incorrecto del día/mes puede ser un problema real. Hay dos soluciones:

- Siempre indica la misma cultura explícita al dar formato y analizar (por ejemplo, la cultura invariante).
- Da formato a DateTime y DateTimeOffsets de una manera independiente de la cultura.

El segundo enfoque es más robusto, especialmente si eliges un formato que coloque el año de cuatro dígitos primero: esas cadenas son mucho más difíciles de interpretar incorrectamente por otra persona. Además, las cadenas formateadas con un formato compatible con los estándares que coloca el año primero (como "o") pueden interpretarse correctamente junto con cadenas formateadas localmente, como un "donante universal". (Las fechas formateadas con "s" o "u" tienen la ventaja adicional de poder ser ordenadas).

Para ilustrar, supongamos que generamos una cadena DateTime insensible a la cultura "s" de la siguiente manera:

```
string s = DateTime.Now.ToString ("o");
```



La cadena de formato "o" incluye los milisegundos en la salida.

La siguiente cadena de formato personalizada proporciona el mismo resultado que "o", pero sin los milisegundos:

```
yyyy-MM-ddTHH:mm:ss K
```

Podemos volver a analizar esto de dos formas. ParseExact exige un cumplimiento estricto con la cadena de formato especificada:

```
DateTime dt1 = DateTime.ParseExact (s, "o", null);
```

(También puedes lograr un resultado similar con los métodos ToString y ToDateTime de XmlConvert.)

Sin embargo, Parse acepta implícitamente tanto el formato "o" como el formato CurrentCulture:

```
DateTime dt2 = DateTime.Parse (s);
```

Esto funciona tanto con DateTime como con DateTimeOffset.



Por lo general, es preferible utilizar ParseExact si conoces el formato de la cadena que estás analizando. Esto significa que si la cadena tiene un formato incorrecto, se lanzará una excepción, lo cual suele ser mejor que correr el riesgo de tener una fecha mal interpretada.

DateTimeStyles

DateTimeStyles es un enum de banderas que proporciona instrucciones adicionales al llamar a Parse en un DateTime(Offset). Aquí están sus miembros:

```
None,
AllowLeadingWhite, AllowTrailingWhite, AllowInnerWhite,
AssumeLocal, AssumeUniversal, AdjustToUniversal,
NoCurrentDateDefault, RoundTripKind
```

También hay un miembro compuesto, AllowWhiteSpaces:

```
AllowWhiteSpaces = AllowLeadingWhite | AllowTrailingWhite | AllowInnerWhite
```

El valor predeterminado es None. Esto significa que normalmente se prohíbe el espacio en blanco adicional (a excepción del espacio en blanco que forma parte de un patrón DateTime estándar).

AssumeLocal y AssumeUniversal se aplican si la cadena no tiene un sufijo de zona horaria (como Z o +9:00). AdjustToUniversal aún respeta los sufijos de zona horaria, pero luego convierte a UTC utilizando la configuración regional actual.

Si analizas una cadena que consta de una hora pero no de una fecha, por defecto se aplica la fecha de hoy. Sin embargo, si aplicas la bandera NoCurrentDateDefault, en su lugar se utilizará el 1 de enero del año 0001.

Formato de Enums

En "Enums" en la página 140, describimos el formato y análisis de los valores enum.

La Tabla 6-6 muestra cada cadena de formato y el resultado de aplicarla a la siguiente expresión:

```
Console.WriteLine (System.ConsoleColor.Red.ToString (formatString));
```

Table 6-6. Enum format strings

Format string	Meaning	Sample output	Notes
G or g	"General"	Red	Default
F or f	Treat as though `Flags` attribute were present	Red	Works on combined members even if `enum` has no `Flags` attribute
D or d	Decimal value	12	Retrieves underlying integral value
X or x	Hexadecimal value	0000000C	Retrieves underlying integral value

Otros mecanismos de conversión

En las secciones anteriores, hemos cubierto los proveedores de formato, que son el mecanismo principal de .NET para el formato y análisis. Sin embargo, existen otros mecanismos de conversión importantes dispersos en varios tipos y espacios de nombres. Algunos realizan conversiones hacia y desde cadenas, y otros realizan otros tipos de conversiones. En esta sección, discutiremos los siguientes temas:

- La clase Convert y sus funciones:
  - ❖ Conversiones de números reales a enteros que redondean en lugar de truncar.
  - ❖ Análisis de números en base 2, 8 y 16.
  - ❖ Conversiones dinámicas.
  - ❖ Traducciones en base 64.
- XmlConvert y su papel en el formato y análisis para XML.
- Los convertidores de tipos y su papel en el formato y análisis para diseñadores y XAML.
- BitConverter, para conversiones binarias.

Convert

.NET llama a los siguientes tipos tipos base:

- bool, char, string, System.DateTime y System.DateTimeOffset
- Todos los tipos numéricos de C#

La clase estática Convert define métodos para convertir cada tipo base a cualquier otro tipo base. Desafortunadamente, la mayoría de estos métodos son inútiles: o lanzan excepciones o son redundantes en comparación con las conversiones implícitas. Sin embargo, entre el desorden, hay algunos métodos útiles, que se enumeran en las siguientes secciones.



Todos los tipos base implementan (de forma explícita) IConvertible, que define métodos para convertir a cada otro tipo base. En la mayoría de los casos, la implementación de cada uno de estos métodos simplemente llama a un método en Convert. En ocasiones raras, puede ser útil escribir un método que acepte un argumento de tipo IConvertible.

Redondeo de conversiones de números reales a enteros

En el Capítulo 2, vimos cómo las conversiones implícitas y explícitas te permiten convertir entre tipos numéricos. En resumen:

- Las conversiones implícitas funcionan para conversiones sin pérdida (por ejemplo, de int a double).
- Las conversiones explícitas son necesarias para conversiones con pérdida (por ejemplo, de double a int).

Las conversiones están optimizadas para la eficiencia; por lo tanto, truncan los datos que no caben. Esto puede ser un problema al convertir de un número real a un entero, ya que a menudo deseas redondear en lugar de truncar. Los métodos de conversión numérica de Convert abordan precisamente este problema, ya que siempre redondean:

```
double d = 3.9;
int i = Convert.ToInt32 (d);    // i == 4
```

Convert utiliza el redondeo del banquero, que ajusta los valores intermedios a enteros pares (esto evita un sesgo positivo o negativo). Si el redondeo del banquero es un problema, primero llama a Math.Round en el número real: esto acepta un argumento adicional que te permite controlar el redondeo de los valores intermedios.

Análisis de números en base 2, 8 y 16

Entre los métodos To(tipo-integral) hay sobrecargas que analizan números en otra base:

```
int thirty = Convert.ToInt32  ("1E", 16);    // Parse in hexadecimal
uint five  = Convert.ToUInt32 ("101", 2);    // Parse in binary
```

El segundo argumento especifica la base. Puede ser cualquier base que desees, ¡siempre y cuando sea 2, 8, 10 o 16!

Conversiones dinámicas

Ocasionalmente, necesitas convertir de un tipo a otro, pero no conoces los tipos hasta que se ejecute el programa. Para esto, la clase Convert proporciona un método ChangeType:

```
public static object ChangeType (object value, Type conversionType);
```

Los tipos de origen y destino deben ser uno de los tipos "base". ChangeType también acepta un argumento opcional IFormatProvider. Aquí tienes un ejemplo:

```
Type targetType = typeof (int);
object source = "42";

object result = Convert.ChangeType (source, targetType);

Console.WriteLine (result);             // 42
Console.WriteLine (result.GetType());   // System.Int32
```

Un ejemplo de cuándo esto puede ser útil es al escribir un deserializador que pueda funcionar con varios tipos. También puede convertir cualquier enumeración a su tipo integral (consultar "Enumeraciones" en la página 140).

Una limitación de ChangeType es que no se puede especificar una cadena de formato o una bandera de análisis.

Conversiones Base-64

A veces, es necesario incluir datos binarios, como un mapa de bits, dentro de un documento de texto, como un archivo XML o un mensaje de correo electrónico. Base 64 es un medio omnipresente para codificar datos binarios como caracteres legibles, utilizando 64 caracteres del conjunto ASCII. El método ToBase64String de Convert convierte un arreglo de bytes a base 64; FromBase64String hace lo contrario.

XmlConvert

Si estás trabajando con datos que se originan en un archivo XML o están destinados para él, XmlConvert (en el espacio de nombres System.Xml) proporciona los métodos más adecuados para el formateo y análisis. Los métodos en XmlConvert manejan los matices del formato XML sin necesidad de formatos especiales. Por ejemplo, true en XML es "true" y no "True". El BCL (Base Class Library) de .NET utiliza internamente XmlConvert de manera extensiva. XmlConvert también es útil para la serialización independiente de la cultura general.

Los métodos de formato en XmlConvert se proporcionan como sobrecargas de los métodos ToString, mientras que los métodos de análisis se llaman ToBoolean, ToDateTime, y así sucesivamente:

```
string s = XmlConvert.ToString (true);          // s = "true"
bool isTrue = XmlConvert.ToBoolean (s);
```

Los métodos que convierten hacia y desde DateTime aceptan un argumento XmlDateTimeSerializationMode. Este es un enumerado con los siguientes valores:

```
Unspecified, Local, Utc, RoundtripKind
```

Local y Utc causan una conversión al formatear (si el DateTime no se encuentra en esa zona horaria). Luego, se agrega la zona horaria a la cadena:

```
2010-02-22T14:08:30.9375          // Unspecified
2010-02-22T14:07:30.9375+09:00    // Local
2010-02-22T05:08:30.9375Z         // Utc
```

Unspecified elimina cualquier información de zona horaria incrustada en el DateTime (es decir, DateTimeKind) antes del formateo. RoundtripKind respeta DateTimeKind del DateTime, por lo que cuando se vuelve a analizar, la estructura DateTime resultante será exactamente igual que la original.

Convertidores de tipo

Los convertidores de tipo (Type Converters) están diseñados para formatear y analizar en entornos de diseño. También analizan valores en documentos Extensible Application Markup Language (XAML), como los utilizados en Windows Presentation Foundation (WPF).

En .NET, existen más de 100 convertidores de tipo, que cubren cosas como colores, imágenes y URIs. En contraste, los proveedores de formato están implementados solo para un puñado de tipos de valores simples.

Los convertidores de tipo suelen analizar cadenas de varias formas, sin necesidad de pistas. Por ejemplo, en una aplicación WPF en Visual Studio, si asignas a un control un color de fondo escribiendo "Beige" en la ventana de propiedades correspondiente, el convertidor de tipo de Color se da cuenta de que te estás refiriendo a un nombre de color y no a una cadena RGB o un color del sistema. Esta flexibilidad a veces hace que los convertidores de tipo sean útiles en contextos fuera de los diseñadores y documentos XAML.

Todos los convertidores de tipo son subclases de TypeConverter en el espacio de nombres System.ComponentModel. Para obtener un TypeConverter, se llama a TypeDescriptor.GetConverter. A continuación, se obtiene un convertidor de tipo para el tipo Color (en el espacio de nombres System.Drawing):

```
TypeConverter cc = TypeDescriptor.GetConverter (typeof (Color));
```

Entre muchos otros métodos, TypeConverter define métodos ConvertToString y ConvertFromString. Podemos llamarlos de la siguiente manera:

```
Color beige  = (Color) cc.ConvertFromString ("Beige");
Color purple = (Color) cc.ConvertFromString ("#800080");
Color window = (Color) cc.ConvertFromString ("Window");
```

Por convención, los convertidores de tipo tienen nombres que terminan en Converter y suelen estar en el mismo espacio de nombres que el tipo que están convirtiendo. Un tipo se vincula a su convertidor mediante un TypeConverterAttribute, lo que permite que los diseñadores recojan automáticamente los convertidores.

Los convertidores de tipo también pueden proporcionar servicios en tiempo de diseño, como generar listas de valores estándar para poblar una lista desplegable en un diseñador o ayudar con la serialización de código.

BitConverter

La mayoría de los tipos base se pueden convertir a una matriz de bytes llamando a BitConverter.GetBytes:

```
foreach (byte b in BitConverter.GetBytes (3.5))
  Console.Write (b + " ");                          // 0 0 0 0 0 0 12 64
```

BitConverter también proporciona métodos, como ToDouble, para convertir en la dirección opuesta.

Los tipos decimal y DateTime(Offset) no son compatibles con BitConverter. Sin embargo, puedes convertir un decimal a una matriz de enteros llamando a decimal.GetBits. Para hacerlo en la dirección opuesta, decimal proporciona un constructor que acepta una matriz de enteros.

En el caso de DateTime, puedes llamar a ToBinary en una instancia, lo cual devuelve un número largo (sobre el cual luego puedes usar BitConverter). El método estático DateTime.FromBinary realiza la operación inversa.

Globalización

Hay dos aspectos en la internacionalización de una aplicación: globalización y localización.

La globalización se ocupa de tres tareas (en orden decreciente de importancia):

- Asegurarse de que tu programa no se rompa al ejecutarse en otra cultura.
- Respetar las reglas de formato de una cultura local, por ejemplo, al mostrar fechas.
- Diseñar tu programa de manera que pueda obtener datos y cadenas específicos de la cultura de ensamblados satélite que puedas escribir y distribuir posteriormente.

La localización implica completar la última tarea escribiendo ensamblados satélite para culturas específicas. Puedes hacer esto después de escribir tu programa (se cubren los detalles en "Recursos y ensamblados satélite" en la página 752).

.NET te ayuda con la segunda tarea al aplicar reglas específicas de la cultura de forma predeterminada. Ya hemos visto cómo llamar a ToString en un objeto DateTime o número respeta las reglas de formato locales. Desafortunadamente, esto hace que sea fácil no cumplir la primera tarea y que tu programa falle porque esperas que las fechas o números se formateen según una cultura asumida. La solución, como hemos visto, es especificar una cultura (como la cultura invariante) al formatear y analizar, o utilizar métodos independientes de la cultura, como los de XmlConvert.

Lista de verificación para la globalización

Ya hemos cubierto los puntos importantes en este capítulo. Aquí tienes un resumen del trabajo esencial requerido:

- Comprender Unicode y las codificaciones de texto (consultar "Codificaciones de texto y Unicode" en la página 281).
- Tener en cuenta que los métodos como ToUpper y ToLower en char y string son sensibles a la cultura: utilizar ToUpperInvariant/ToLowerInvariant a menos que se desee sensibilidad a la cultura.
- Favorecer los mecanismos de formato y análisis independientes de la cultura para DateTime y DateTimeOffset, como ToString("o") y XmlConvert.
- De lo contrario, especificar una cultura al formatear/analisar números o fechas/horas (a menos que se desee el comportamiento de la cultura local).

Pruebas

Puedes realizar pruebas con diferentes culturas al reasignar la propiedad CurrentCulture del objeto Thread (en System.Threading). El siguiente ejemplo cambia la cultura actual a Turquía:

```
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo ("tr-TR");
```

Turquía es un caso de prueba especialmente interesante porque:

- "i".ToUpper() != "I" y "I".ToLower() != "i".
- Las fechas se formatean como día.mes.año (separadas por puntos).
- El separador decimal es una coma en lugar de un punto.

También puedes experimentar cambiando la configuración de formato de números y fechas en el Panel de control de Windows: esto se reflejará en la cultura predeterminada (CultureInfo.CurrentCulture).

CultureInfo.GetCultures() devuelve una matriz de todas las culturas disponibles.



El objeto Thread y CultureInfo también admiten una propiedad CurrentUICulture. Esta se relaciona más con la localización, que se aborda en el Capítulo 17.

Trabajando con números

Conversiones

Ya hemos cubierto las conversiones numéricas en capítulos y secciones anteriores; la Tabla 6-7 resume todas las opciones.

Table 6-7. Summary of numeric conversions

Task	Functions	Examples

e	`Parse`
	`TryParse`
		`double d = double.Parse ("3.5");`
		`int i;`
		`bool ok = int.TryParse ("3", out i);`

Parsing from base 2, 8, or 16	`Convert.ToIntegral`	`int i = Convert.ToInt32 ("1E", 16);`
Formatting to hexadecimal	`ToString ("X")`	`string hex = 45.ToString ("X");`
Lossless numeric conversion	Implicit cast	`int i = 23;` `double d = i;`
Truncating numeric conversion	Explicit cast	`double d = 23.5;` `int i = (int) d;`

Task	Functions	Examples
Rounding numeric conversion (real to integral)	`Convert.ToIntegral`	`double d = 23.5;` `int i = Convert.ToInt32 (d);`

Matemáticas

La Tabla 6-8 enumera los miembros clave de la clase estática Math. Las funciones trigonométricas aceptan argumentos de tipo double; otros métodos como Max están sobrecargados para operar en todos los tipos numéricos. La clase Math también define las constantes matemáticas E (e) y PI.

Table 6-8. Methods in the static Math class

Category	Methods
Rounding	`Round, Truncate, Floor, Ceiling`
Maximum/minimum	`Max, Min`
Absolute value and sign	`Abs, Sign`
Square root	`Sqrt`
Raising to a power	`Pow, Exp`
Logarithm	`Log, Log10`

Trigonometric	`Sin, Cos, Tan,`
	`Sinh, Cosh, Tanh,`
	`Asin, Acos, Atan`

El método Round te permite especificar el número de decimales con los que redondear, así como cómo manejar los puntos medios (hacia arriba o con el redondeo bancario). Floor y Ceiling redondean al entero más cercano: Floor siempre redondea hacia abajo, y Ceiling siempre redondea hacia arriba, incluso con números negativos.

Max y Min aceptan solo dos argumentos. Si tienes un array o una secuencia de números, utiliza los métodos de extensión Max y Min en System.Linq.Enumerable.

BigInteger

La estructura BigInteger es un tipo numérico especializado. Se encuentra en el espacio de nombres System.Numerics y te permite representar enteros arbitrariamente grandes sin pérdida de precisión.

C# no proporciona soporte nativo para BigInteger, por lo que no hay forma de representar literales de BigInteger. Sin embargo, puedes convertir implícitamente cualquier otro tipo integral a un BigInteger:

```
BigInteger twentyFive = 25;      // implicit conversion from integer
```

Para representar un número más grande, como un "googol" (10^100), puedes usar uno de los métodos estáticos de BigInteger, como Pow (elevar a la potencia):

```
BigInteger googol = BigInteger.Pow (10, 100);
```

Alternativamente, puedes usar Parse para convertir desde una cadena:

```
BigInteger googol = BigInteger.Parse ("1".PadRight (101, '0'));
```

Llamar a ToString() en este caso imprime cada dígito:

```
Console.WriteLine (googol.ToString()); // 10000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Puedes realizar conversiones potencialmente con pérdida de precisión entre BigInteger y los tipos numéricos estándar utilizando el operador de conversión explícita:

```
double g2 = (double) googol;        // Explicit cast
BigInteger g3 = (BigInteger) g2;    // Explicit cast
Console.WriteLine (g3);
```

La salida de esto demuestra la pérdida de precisión:

```
99999999999999996733616880411669l2...
```

BigInteger sobrecarga todos los operadores aritméticos, incluyendo el resto (%), así como los operadores de comparación e igualdad.

También puedes construir un BigInteger a partir de un array de bytes. El siguiente código genera un número aleatorio de 32 bytes adecuado para criptografía y luego lo asigna a un BigInteger:

```
// This uses the System.Security.Cryptography namespace:
RandomNumberGenerator rand = RandomNumberGenerator.Create();
byte[] bytes = new byte [32];
rand.GetBytes (bytes);
var bigRandomNumber = new BigInteger (bytes);   // Convert to BigInteger
```

La ventaja de almacenar dicho número en un BigInteger en lugar de un array de bytes es que obtienes semántica de tipo valor. Llamar a ToByteArray convierte un BigInteger nuevamente a un array de bytes.

Half

La estructura Half es un tipo de punto flotante de 16 bits y fue introducida con .NET 5. Half se utiliza principalmente para interoperar con procesadores de tarjetas gráficas y no tiene soporte nativo en la mayoría de las CPU.

Puedes convertir entre Half y float o double mediante una conversión explícita:

```
Half h = (Half) 123.456;
Console.WriteLine (h);     // 123.44  (note loss of precision)
```

No se definen operaciones aritméticas para este tipo, por lo que debes convertirlo a otro tipo, como float o double, para realizar cálculos.

Half tiene un rango de -65500 a 65500:

```
Console.WriteLine (Half.MinValue);   // -65500
Console.WriteLine (Half.MaxValue);   // 65500
```

Ten en cuenta la pérdida de precisión en el rango máximo:

```
Console.WriteLine ((Half)65500);     // 65500
Console.WriteLine ((Half)65490);     // 65500
Console.WriteLine ((Half)65480);     // 65470
```

Complex

La estructura Complex es otro tipo numérico especializado que representa números complejos con componentes reales e imaginarias de tipo double. Complex reside en el espacio de nombres (junto con BigInteger).

Para usar Complex, instancia la estructura especificando los valores reales e imaginarios:

```
var c1 = new Complex (2, 3.5);
var c2 = new Complex (3, 0);
```

También hay conversiones implícitas desde los tipos numéricos estándar.

La estructura Complex expone propiedades para los valores reales e imaginarios, así como para la fase y magnitud:

```
Console.WriteLine (c1.Real);       // 2
Console.WriteLine (c1.Imaginary);  // 3.5
Console.WriteLine (c1.Phase);      // 1.05165021254837
Console.WriteLine (c1.Magnitude);  // 4.03112887414927
```

También puedes construir un número complejo especificando la magnitud y la fase:

```
Complex c3 = Complex.FromPolarCoordinates (1.3, 5);
```

Los operadores aritméticos estándar están sobrecargados para funcionar con números complejos:

```
Console.WriteLine (c1 + c2);    // (5, 3.5)
Console.WriteLine (c1 * c2);    // (6, 10.5)
```

La estructura Complex expone métodos estáticos para funciones más avanzadas, que incluyen lo siguiente:

- Trigonométricas (Sin, Asin, Sinh, Tan, etc.)
- Logaritmos y exponenciales
- Conjugado

Aleatorio (Random)

La clase Random genera una secuencia pseudorandom de bytes, enteros o números decimales (doubles).

Para utilizar Random, primero debes instanciarlo, proporcionando opcionalmente una semilla (seed) para iniciar la serie de números random. Al utilizar la misma semilla se garantiza la misma secuencia de números (si se ejecuta bajo la misma versión del CLR), lo cual puede ser útil cuando se desea reproducibilidad:

```
Random r1 = new Random (1);
Random r2 = new Random (1);
Console.WriteLine (r1.Next (100) + ", " + r1.Next (100));      // 24, 11
Console.WriteLine (r2.Next (100) + ", " + r2.Next (100));      // 24, 11
```

Si no deseas reproducibilidad, puedes construir un objeto Random sin semilla; en ese caso, utiliza el tiempo del sistema actual para generar una.



Debido a que el reloj del sistema tiene una granularidad limitada, dos instancias de Random creadas muy cerca una de la otra (normalmente dentro de los 10 ms) producirán la misma secuencia de valores. Una trampa común es instanciar un nuevo objeto Random cada vez que necesitas un número aleatorio en lugar de reutilizar el mismo objeto.

Un buen patrón es declarar una única instancia estática de Random. Sin embargo, en escenarios de programación multihilo, esto puede causar problemas debido a que los objetos Random no son seguros para subprocesos. Describimos una solución alternativa en "Almacenamiento local en subprocesos" en la página 898.

Llamar a Next(n) genera un número entero aleatorio entre 0 y n-1. NextDouble genera un número decimal aleatorio entre 0 y 1. NextBytes llena una matriz de bytes con valores aleatorios.

Random no se considera lo suficientemente aleatorio para aplicaciones de alta seguridad como la criptografía. Para esto, .NET proporciona un generador de números aleatorios criptográficamente seguro en el espacio de nombres System.Security.Cryptography. Así es cómo se utiliza:

```
var rand = System.Security.Cryptography.RandomNumberGenerator.Create();
byte[] bytes = new byte [32];
rand.GetBytes (bytes);       // Fill the byte array with random numbers.
```

La desventaja es que es menos flexible: llenar una matriz de bytes es el único medio para obtener números aleatorios. Para obtener un número entero, debes usar BitConverter:

```
byte[] bytes = new byte [4];
rand.GetBytes (bytes);
int i = BitConverter.ToInt32 (bytes, 0);
```

BitOperations

La clase System.Numerics.BitOperations (a partir de .NET 6) expone los siguientes métodos para ayudar con operaciones en base-2:

IsPow2

Devuelve true si un número es una potencia de 2.

LeadingZeroCount/TrailingZeroCount

Devuelve el número de ceros principales o finales, cuando se formatea como un entero sin signo de 32 o 64 bits en base-2.

Log2

Devuelve el logaritmo base-2 entero de un número entero sin signo.

PopCount

Devuelve el número de bits establecidos en 1 en un número entero sin signo.

RotateLeft/RotateRight

Realiza una rotación de bits a la izquierda o a la derecha.

RoundUpToPowerOf2

Redondea un número entero sin signo al siguiente número entero más cercano que sea una potencia de 2.

Enums

En el Capítulo 3, describimos el tipo enum de C# y mostramos cómo combinar miembros, probar igualdad, usar operadores lógicos y realizar conversiones. .NET amplía el soporte de C# para los enums a través del tipo System.Enum. Este tipo tiene dos roles:

- Proporcionar unificación de tipo para todos los tipos enum.
- Definir métodos de utilidad estáticos.

La unificación de tipo significa que puedes convertir implícitamente cualquier miembro de un enum a una instancia de System.Enum:

```
Display (Nut.Macadamia);     // Nut.Macadamia
Display (Size.Large);        // Size.Large

void Display (Enum value)
{
  Console.WriteLine (value.GetType().Name + "." + value.ToString());
}

enum Nut  { Walnut, Hazelnut, Macadamia }
enum Size { Small, Medium, Large }
```

Los métodos de utilidad estáticos en System.Enum están principalmente relacionados con realizar conversiones y obtener listas de miembros.

Conversiones de Enum

Existen tres formas de representar un valor de enum:

- Como miembro del enum.
- Como su valor integral subyacente.
- Como una cadena de texto.

En esta sección, describiremos cómo realizar conversiones entre cada una de ellas.

Conversiones de enum a integral

Recuerda que una conversión explícita convierte entre un miembro del enum y su valor integral. Una conversión explícita es el enfoque correcto si conoces el tipo enum en tiempo de compilación:

```
[Flags]
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
int i = (int) BorderSides.Top;             // i == 4
BorderSides side = (BorderSides) i;        // side == BorderSides.Top
```

También puedes convertir una instancia de System.Enum a su tipo integral de la misma manera. El truco consiste en realizar primero una conversión a object y luego al tipo integral:

```
static int GetIntegralValue (Enum anyEnum)
{
  return (int) (object) anyEnum;
}
```

Esto se basa en que conozcas el tipo integral. El método que acabamos de escribir generaría un error si se le pasara un enum cuyo tipo integral fuera long. Para escribir un método que funcione con un enum de cualquier tipo integral, puedes seguir una de estas tres aproximaciones. La primera es llamar a Convert.ToDecimal:

```
static decimal GetAnyIntegralValue (Enum anyEnum)
{
  return Convert.ToDecimal (anyEnum);
}
```

Esto funciona porque todos los tipos integrales (incluido ulong) se pueden convertir a decimal sin pérdida de información. La segunda aproximación es llamar a Enum.GetUnderlyingType para obtener el tipo integral del enum y luego llamar a Convert.ChangeType:

```
static object GetBoxedIntegralValue (Enum anyEnum)
{
  Type integralType = Enum.GetUnderlyingType (anyEnum.GetType());
  return Convert.ChangeType (anyEnum, integralType);
}
```

Esto preserva el tipo integral original, como se muestra en el siguiente ejemplo:

```
object result = GetBoxedIntegralValue (BorderSides.Top);
Console.WriteLine (result);                              // 4
Console.WriteLine (result.GetType());                    // System.Int32
```



Nuestro método GetBoxedIntegralValue de hecho no realiza ninguna conversión de valores; en cambio, coloca el mismo valor en otro tipo. Traduce un valor integral vestido como tipo enum a un valor integral vestido como tipo integral. Explicamos esto con más detalle en "Cómo funcionan los enums" en la página 323.

La tercera aproximación consiste en llamar a Format o ToString especificando la cadena de formato "d" o "D". Esto te proporciona el valor integral del enum como una cadena y es útil al escribir formateadores de serialización personalizados:

```
static string GetIntegralValueAsString (Enum anyEnum)
{
  return anyEnum.ToString ("D");      // returns something like "4"
}
```

Conversiones de integral a enum

Enum.ToObject convierte un valor integral en una instancia de enum del tipo especificado:

```
object bs = Enum.ToObject (typeof (BorderSides), 3);
Console.WriteLine (bs);                              // Left, Right
```

Esto es equivalente a lo siguiente:

```
BorderSides bs = (BorderSides) 3;
```

ToObject está sobrecargado para aceptar todos los tipos integrales, así como object (este último funciona con cualquier tipo integral encajado).

Conversiones de string

Para convertir un enum a una cadena, puedes llamar al método estático Enum.Format o llamar a ToString en la instancia. Cada método acepta una cadena de formato que puede ser "G" para el comportamiento de formato predeterminado, "D" para emitir el valor integral subyacente como una cadena, "X" para lo mismo en hexadecimal o "F" para formatear los miembros combinados de un enum sin el atributo Flags. Enumeramos ejemplos de estos en "Cadenas de formato estándar y banderas de análisis" en la página 303.

Enum.Parse convierte una cadena en un enum. Acepta el tipo de enum y una cadena que puede incluir varios miembros:

```
BorderSides leftRight = (BorderSides) Enum.Parse (typeof (BorderSides),
                                                  "Left, Right");
```

Un tercer argumento opcional te permite realizar un análisis sin distinguir mayúsculas y minúsculas. Se lanza una excepción de ArgumentException si no se encuentra el miembro.

Enumeración de valores de enum

Enum.GetValues devuelve un array que contiene todos los miembros de un tipo de enum específico:

```
foreach (Enum value in Enum.GetValues (typeof (BorderSides)))
  Console.WriteLine (value);
```

Los miembros compuestos como "LeftRight = Left | Right" también están incluidos. Enum.GetNames realiza la misma función pero devuelve un arreglo de cadenas.



Internamente, el CLR implementa GetValues y GetNames mediante la reflexión de los campos en el tipo de enumeración. Los resultados se almacenan en caché para mayor eficiencia.

Cómo funcionan las enumeraciones

Las reglas semánticas de las enumeraciones son aplicadas en gran medida por el compilador. En el CLR, no hay diferencia en tiempo de ejecución entre una instancia de enumeración (cuando no está en formato de caja) y su valor integral subyacente. Además, una definición de enumeración en el CLR es simplemente un subtipo de System.Enum con campos estáticos de tipo integral para cada miembro. Esto hace que el uso ordinario de una enumeración sea altamente eficiente, con un costo en tiempo de ejecución similar al de las constantes enteras.

El inconveniente de esta estrategia es que las enumeraciones proporcionan seguridad de tipo estática pero no fuerte. Vimos un ejemplo de esto en el Capítulo 3:

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
BorderSides b = BorderSides.Left;
b += 1234;                          // No error!
```

Cuando el compilador no puede realizar la validación (como en este ejemplo), no hay respaldo por parte del tiempo de ejecución para lanzar una excepción.

Lo que dijimos acerca de que no hay diferencia en tiempo de ejecución entre una instancia de enumeración y su valor integral puede parecer contradictorio con lo siguiente:

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
Console.WriteLine (BorderSides.Right.ToString());        // Right
Console.WriteLine (BorderSides.Right.GetType().Name);    // BorderSides
```

Dada la naturaleza de una instancia de enumeración en tiempo de ejecución, esperarías que esto imprima 2 e Int32. La razón de este comportamiento se debe a algunas manipulaciones en tiempo de compilación. C# explícitamente realiza una operación de encajonamiento (boxing) de una instancia de enumeración antes de llamar a sus métodos virtuales, como ToString o GetType. Y cuando una instancia de enumeración es encajonada, adquiere un contenedor en tiempo de ejecución que hace referencia a su tipo de enumeración.

La estructura Guid

La estructura Guid representa un identificador único a nivel mundial: un valor de 16 bytes que, al generarse, es casi seguro que es único en el mundo. Los Guid se utilizan a menudo como claves en diversas aplicaciones y bases de datos. Hay 2^128 o 3.4 × 10^38 Guids únicos.

El método estático Guid.NewGuid genera un Guid único:

```
Guid g = Guid.NewGuid ();
Console.WriteLine (g.ToString());  // 0d57629c-7d6e-4847-97cb-9e2fc25083fe
```

Para instanciar un valor existente, se utilizan uno de los constructores. Los dos constructores más útiles son:

```
public Guid (byte[] b);    // Accepts a 16-byte array
public Guid (string g);    // Accepts a formatted string
```

Cuando se representa como una cadena, un Guid se formatea como un número hexadecimal de 32 dígitos, con guiones opcionales después del 8º, 12º, 16º y 20º dígito. Toda la cadena también puede estar opcionalmente encerrada entre corchetes o llaves:

```
Guid g1 = new Guid ("{0d57629c-7d6e-4847-97cb-9e2fc25083fe}");
Guid g2 = new Guid ("0d57629c7d6e484797cb9e2fc25083fe");
Console.WriteLine (g1 == g2);  // True
```

Siendo una estructura (struct), un Guid sigue las reglas de semántica de tipos de valor; por lo tanto, el operador de igualdad funciona en el ejemplo anterior.

El método ToByteArray convierte un Guid en un arreglo de bytes.

La propiedad estática Guid.Empty devuelve un Guid vacío (todos los ceros). Esto se utiliza a menudo en lugar de null.

Comparación de igualdad

Hasta ahora, hemos asumido que los operadores == y != son todo lo que se necesita para la comparación de igualdad. Sin embargo, la cuestión de la igualdad es más compleja y sutil, a veces requiriendo el uso de métodos e interfaces adicionales. Esta sección explora los protocolos estándar de C# y .NET para la igualdad, centrándose especialmente en dos preguntas:

- ¿Cuándo son adecuados (y cuándo no) los operadores == y != para la comparación de igualdad, y cuáles son las alternativas?
- ¿Cómo y cuándo se debe personalizar la lógica de igualdad de un tipo?

Pero antes de explorar los detalles de los protocolos de igualdad y cómo personalizarlos, primero debemos analizar el concepto preliminar de igualdad de valor frente a igualdad referencial.

Igualdad de valor frente a igualdad referencial

Existen dos tipos de igualdad:

Igualdad de valor

Dos valores son equivalentes en algún sentido.

Igualdad referencial

Dos referencias se refieren exactamente al mismo objeto.

A menos que se anule:

- Los tipos de valor utilizan la igualdad de valor.
- Los tipos de referencia utilizan la igualdad referencial. (Esto se anula con tipos anónimos y registros.)

De hecho, los tipos de valor solo pueden utilizar la igualdad de valor (a menos que estén en una caja). Una demostración sencilla de igualdad de valor es comparar dos números:

```
int x = 5, y = 5;
Console.WriteLine (x == y);   // True (by virtue of value equality)
```

Una demostración más elaborada es comparar dos estructuras DateTimeOffset. Lo siguiente imprime True porque los dos DateTimeOffset se refieren al mismo punto en el tiempo y, por lo tanto, se consideran equivalentes:

```
var dt1 = new DateTimeOffset (2010, 1, 1, 1, 1, 1, TimeSpan.FromHours(8));
var dt2 = new DateTimeOffset (2010, 1, 1, 2, 1, 1, TimeSpan.FromHours(9));
Console.WriteLine (dt1 == dt2);   // True
```



DateTimeOffset es una estructura cuyas semánticas de igualdad han sido ajustadas. Por defecto, las estructuras exhiben un tipo especial de igualdad de valor llamada igualdad estructural en la que dos valores se consideran iguales si todos sus miembros son iguales. (Puede ver esto creando una estructura y llamando a su método Equals; más sobre esto más adelante.)

Los tipos de referencia exhiben la igualdad referencial por defecto. En el siguiente ejemplo, f1 y f2 no son iguales a pesar de que sus objetos tienen contenido idéntico:

```
class Foo { public int X; }
...
Foo f1 = new Foo { X = 5 };
Foo f2 = new Foo { X = 5 };
Console.WriteLine (f1 == f2);   // False
```

En contraste, f3 y f1 son iguales porque hacen referencia al mismo objeto:

```
Foo f3 = f1;
Console.WriteLine (f1 == f3);   // True
```

Más adelante en esta sección, explicaremos cómo personalizar los tipos de referencia para exhibir igualdad de valor. Un ejemplo de esto es la clase Uri en el espacio de nombres System:

```
Uri uri1 = new Uri ("http://www.linqpad.net");
Uri uri2 = new Uri ("http://www.linqpad.net");
Console.WriteLine (uri1 == uri2);              // True
```

La clase string exhibe un comportamiento similar:

```
var s1 = "http://www.linqpad.net";
var s2 = "http://" + "www.linqpad.net";
Console.WriteLine (s1 == s2);       // True
```

Protocolos estándar de igualdad

Existen tres protocolos estándar que los tipos pueden implementar para la comparación de igualdad:

- Los operadores == and !=
- El método virtual Equals en object
- La interfaz IEquatable<T>

Además, existen protocolos enchufables y la interfaz IStructuralEquatable, que se describe en el Capítulo 7.

== and !=

Ya hemos visto en muchos ejemplos cómo los operadores estándar == and != realizan comparaciones de igualdad/desigualdad. Las sutilezas con == and != surgen porque son operadores; por lo tanto, se resuelven estáticamente (de hecho, se implementan como funciones estáticas). Entonces, cuando usas == or !=, C# toma una decisión en tiempo de compilación sobre qué tipo realizará la comparación y no se tiene en cuenta el comportamiento virtual. Esto es normalmente deseable. En el siguiente ejemplo, el compilador establece == para el tipo int porque x e y son ambos int:

```
int x = 5;
int y = 5;
Console.WriteLine (x == y);      // True
```

Pero en el siguiente ejemplo, el compilador establece el operador == para el tipo object:

```
object x = 5;
object y = 5;
Console.WriteLine (x == y);      // False
```

Como object es una clase (y por lo tanto un tipo de referencia), el operador == de object utiliza la igualdad referencial para comparar x e y. El resultado es falso porque x e y se refieren a objetos en cajas diferentes en el montón.

El método virtual Object.Equals

Para igualar correctamente x e y en el ejemplo anterior, podemos usar el método virtual Equals. Equals está definido en System.Object y, por lo tanto, está disponible para todos los tipos:

```
object x = 5;
object y = 5;
Console.WriteLine (x.Equals (y));      // True
```

Equals se resuelve en tiempo de ejecución, según el tipo real del objeto. En este caso, llama al método Equals de Int32, que aplica la igualdad de valor a los operandos, devolviendo true. Con los tipos de referencia, Equals realiza una comparación de igualdad referencial de forma

predeterminada; con las estructuras, Equals realiza una comparación estructural llamando a Equals en cada uno de sus campos.

Por qué la complejidad?

Es posible que te preguntes por qué los diseñadores de C# no evitaron el problema haciendo que == sea virtual y, por lo tanto, funcionalmente idéntico a Equals. Hay tres razones para esto:

• Si el primer operando es null, Equals falla con una NullReferenceException; un operador estático no lo hace.

• Debido a que el operador == se resuelve estáticamente, se ejecuta extremadamente rápido. Esto significa que puedes escribir código computacionalmente intensivo sin penalización, y sin necesidad de aprender otro lenguaje como C++.

• A veces puede ser útil que == y Equals apliquen diferentes definiciones de igualdad. Describiremos este escenario más adelante en esta sección.

En esencia, la complejidad del diseño refleja la complejidad de la situación: el concepto de igualdad abarca una multitud de escenarios.

Por lo tanto, Equals es adecuado para equilibrar dos objetos de manera agnóstica al tipo. El siguiente método compara dos objetos de cualquier tipo:

```
public static bool AreEqual (object obj1, object obj2)
  => obj1.Equals (obj2);
```

Sin embargo, hay un caso en el que esto falla. Si el primer argumento es null, se produce una NullReferenceException. Aquí está la solución:

```
public static bool AreEqual (object obj1, object obj2)
{
  if (obj1 == null) return obj2 == null;
  return obj1.Equals (obj2);
}
```

O de manera más concisa:

```
public static bool AreEqual (object obj1, object obj2)
  => obj1 == null ? obj2 == null : obj1.Equals (obj2);
```

El método estático object.Equals

La clase object proporciona un método auxiliar estático que realiza el trabajo de AreEqual en el ejemplo anterior. Su nombre es Equals, al igual que el método virtual, pero no hay conflicto porque acepta dos argumentos:

```
public static bool Equals (object objA, object objB)
```

Esto proporciona un algoritmo de comparación de igualdad seguro para null cuando los tipos son desconocidos en tiempo de compilación:

```
object x = 3, y = 3;
Console.WriteLine (object.Equals (x, y));   // True
x = null;
Console.WriteLine (object.Equals (x, y));   // False
y = null;
Console.WriteLine (object.Equals (x, y));   // True
```

Una aplicación útil es al escribir tipos genéricos. El siguiente código no se compilará si object.Equals se reemplaza con el operador == o !=:

```
class Test <T>
{
  T _value;
  public void SetValue (T newValue)
  {
    if (!object.Equals (newValue, _value))
    {
      _value = newValue;
      OnValueChanged();
    }
  }
  protected virtual void OnValueChanged() { ... }
}
```

Los operadores están prohibidos aquí porque el compilador no puede enlazar al método estático de un tipo desconocido.



Una forma más elaborada de implementar esta comparación es con la clase EqualityComparer<T>. Esto tiene la ventaja de evitar el boxing:

```
if (!EqualityComparer<T>.Default.Equals (newValue, _value))
```

Discutimos EqualityComparer<T> en más detalle en el Capítulo 7 (ver "Plugging in Equality and Order" en la página 388).

El método estático object.ReferenceEquals

Ocasionalmente, es posible que necesites forzar una comparación de igualdad referencial. El método estático object.ReferenceEquals hace precisamente eso:

```
Widget w1 = new Widget();
Widget w2 = new Widget();
Console.WriteLine (object.ReferenceEquals (w1, w2));     // False

class Widget { ... }
```

Puedes querer hacer esto porque es posible que Widget anule el método Equals virtual de manera que w1.Equals(w2) devuelva true. Además, es posible que Widget sobrecargue el

operador == de manera que w1==w2 también devuelva true. En tales casos, llamar a object.ReferenceEquals garantiza la semántica normal de igualdad referencial.

 Otra forma de forzar una comparación de igualdad referencial es hacer un cast de los valores a tipo `object` y luego aplicar el operador `==`.

La interfaz `IEquatable<T>`

Una consecuencia de llamar a `object.Equals` es que se realiza un boxing (empaquetado) en los tipos de valor. Esto es indeseable en escenarios donde el rendimiento es crítico, ya que el boxing es relativamente costoso en comparación con la comparación real. Se introdujo una solución en C# 2.0 con la interfaz `IEquatable<T>`:

```
public interface IEquatable<T>
{
  bool Equals (T other);
}
```

La idea es que `IEquatable<T>`, cuando se implementa, produce el mismo resultado que llamar al método virtual `Equals` de `object`, pero de manera más rápida. La mayoría de los tipos básicos de .NET implementan `IEquatable<T>`. Puedes usar `IEquatable<T>` como una restricción en un tipo genérico:

```
class Test<T> where T : IEquatable<T>
{
  public bool IsEqual (T a, T b)
  {
    return a.Equals (b);     // No boxing with generic T
  }
}
```

Si eliminamos la restricción genérica, la clase aún se compilaría, pero `a.Equals(b)` se vincularía al método más lento `object.Equals` (asumiendo que `T` es un tipo de valor).

Cuando `Equals` y `==` no son iguales

Mencionamos anteriormente que a veces es útil que `==` y `Equals` apliquen diferentes definiciones de igualdad. Por ejemplo:

```
double x = double.NaN;
Console.WriteLine (x == x);            // False
Console.WriteLine (x.Equals (x));      // True
```

El operador `==` del tipo `double` establece que un valor `NaN` nunca puede ser igual a cualquier otro valor, incluso otro `NaN`. Esto es más natural desde una perspectiva matemática y refleja el

comportamiento subyacente de la CPU. Sin embargo, el método `Equals` está obligado a aplicar una igualdad reflexiva, es decir:

`x.Equals (x)` must *always* return true.

Las colecciones y diccionarios dependen de que `Equals` se comporte de esta manera; de lo contrario, no podrían encontrar un elemento que se haya almacenado previamente.

En realidad, es bastante raro que `Equals` y `==` apliquen diferentes definiciones de igualdad en tipos de valor. Un escenario más común ocurre con los tipos de referencia, esto sucede cuando el autor personaliza el método `Equals` para que realice una comparación de igualdad de valores, mientras que `==` realiza la igualdad referencial (por defecto). La clase `StringBuilder` hace exactamente eso:

```
var sb1 = new StringBuilder ("foo");
var sb2 = new StringBuilder ("foo");
Console.WriteLine (sb1 == sb2);          // False (referential equality)
Console.WriteLine (sb1.Equals (sb2));    // True  (value equality)
```

Veamos ahora cómo personalizar la igualdad.

Igualdad y tipos personalizados

Recordemos el comportamiento predeterminado de la comparación de igualdad:

- Los tipos de valor utilizan la igualdad de valores.
- Los tipos de referencia utilizan la igualdad referencial a menos que se sobrescriba (como en el caso de los tipos anónimos y los registros).

Además:

- El método `Equals` de una estructura aplica la igualdad de valores estructurales de manera predeterminada (es decir, compara cada campo en la estructura).

En ocasiones, tiene sentido anular este comportamiento al escribir un tipo. Hay dos casos para hacerlo:

- Cambiar el significado de la igualdad.
- Acelerar las comparaciones de igualdad para las estructuras.

Cambiar el significado de la igualdad

Cambiar el significado de la igualdad tiene sentido cuando el comportamiento predeterminado de `==` y `Equals` es antinatural para tu tipo y no es lo que un consumidor esperaría. Un ejemplo es `DateTimeOffset`, una estructura con dos campos privados: un `DateTime` en formato UTC y un desplazamiento numérico entero. Si estuvieras escribiendo este tipo, probablemente querrías asegurarte de que las comparaciones de igualdad solo consideren el campo UTC `DateTime` y no el campo de desplazamiento. Otro ejemplo son los tipos numéricos que admiten valores `NaN`,

como `float` y `double`. Si estuvieras implementando tales tipos, querrías asegurarte de que la lógica de comparación de `NaN` estuviera soportada en las comparaciones de igualdad.

Con las clases, a veces es más natural ofrecer la igualdad de valores como predeterminada en lugar de la igualdad referencial. Esto suele ser cierto para clases pequeñas que contienen un simple dato, como `System.Uri` o `System.String`.

Con los registros, el compilador implementa automáticamente la igualdad estructural (comparando cada campo). Sin embargo, en ocasiones esto incluirá campos que no deseas comparar o objetos que requieren una lógica especial de comparación, como colecciones.

El proceso de anular la igualdad con registros es ligeramente diferente porque los registros siguen un patrón especial diseñado para funcionar bien con las reglas de herencia.

Acelerar las comparaciones de igualdad con estructuras

El algoritmo predeterminado de igualdad estructural para las estructuras es relativamente lento. Tomar el control de este proceso al anular `Equals` puede mejorar el rendimiento en un factor de cinco. Sobrecargar el operador `==` e implementar `IEquatable<T>` permite comparaciones de igualdad sin caja y esto puede acelerar las cosas en un factor de cinco nuevamente.



Anular la semántica de igualdad para tipos de referencia no beneficia el rendimiento. El algoritmo predeterminado para la comparación de igualdad referencial ya es muy rápido porque simplemente compara dos referencias de 32 o 64 bits.

Hay otro caso bastante peculiar para personalizar la igualdad, y es mejorar el algoritmo de hash de una estructura para un mejor rendimiento en una tabla hash. Esto se debe al hecho de que la comparación de igualdad y el hash están íntimamente relacionados. Examinaremos el hash en un momento.

Cómo anular la semántica de igualdad

Para anular la igualdad con clases o estructuras, aquí están los pasos:

1. Anula `GetHashCode()` y `Equals()`.

2. (Opcionalmente) sobrecarga `!=` y `==`.

3. (Opcionalmente) implementa `IEquatable<T>`.

El proceso es diferente (y más simple) con los registros porque el compilador ya anula los métodos y operadores de igualdad de acuerdo con su propio patrón especial. Si deseas intervenir, debes cumplir con este patrón, lo que significa escribir un método `Equals` con una firma como esta:

```
record Test (int X, int Y)
{
  public virtual bool Equals (Test t) => t != null && t.X == X && t.Y == Y;
}
```

Observa que `Equals` es virtual (no override) y acepta el tipo de registro real (`Test` en este caso, no `object`). El compilador reconocerá que tu método tiene la firma "correcta" y lo parcheará.

También debes anular `GetHashCode()`, al igual que lo harías con clases o estructuras. No necesitas (y no debes) sobrecargar `!=` y `==`, ni implementar `IEquatable<T>`, porque esto ya se hace automáticamente por ti.

Anular GetHashCode

Puede parecer extraño que System.Object, con su pequeño conjunto de miembros, defina un método con un propósito especializado y limitado. GetHashCode es un método virtual en Object que se ajusta a esta descripción; existe principalmente para beneficiar únicamente a los dos siguientes tipos:

```
System.Collections.Hashtable
System.Collections.Generic.Dictionary<TKey,TValue>
```

Estos son hashtable (tablas hash): colecciones en las que cada elemento tiene una clave que se utiliza para almacenar y recuperar. Un hashtable aplica una estrategia muy específica para asignar eficientemente elementos en función de su clave. Esto requiere que cada clave tenga un número Int32, o código hash. El código hash no necesita ser único para cada clave, pero debe ser lo más variado posible para obtener un buen rendimiento de hashtable. Los hashtables se consideran lo suficientemente importantes como para que GetHashCode se defina en System.Object, de modo que cada tipo pueda generar un código hash.



Describimos los hashtables en detalle en el Capítulo 7.

Tanto los tipos de referencia como los tipos de valor tienen implementaciones predeterminadas de GetHashCode, lo que significa que no necesitas anular este método, a menos que anules Equals. (Y si anulas GetHashCode, casi seguramente también querrás anular Equals).

Aquí están las otras reglas para anular object.GetHashCode:

- Debe devolver el mismo valor en dos objetos para los cuales Equals devuelve true (por lo tanto, GetHashCode y Equals se anulan juntos).
- No debe generar excepciones.
- Debe devolver el mismo valor si se llama repetidamente en el mismo objeto (a menos que el objeto haya cambiado).

Para obtener el máximo rendimiento en los hashtables, debes escribir GetHashCode de manera que se minimice la probabilidad de que dos valores diferentes devuelvan el mismo código hash. Esto da lugar a la tercera razón para anular Equals y GetHashCode en estructuras, que es proporcionar un algoritmo de hashing más eficiente que el predeterminado. La implementación predeterminada para estructuras queda a discreción del tiempo de ejecución y puede basarse en cada campo de la estructura.

En contraste, la implementación predeterminada de GetHashCode para clases se basa en un token de objeto interno, que es único para cada instancia en la implementación actual del CLR (Common Language Runtime).



Si el código hash de un objeto cambia después de haber sido agregado como clave a un diccionario, el objeto ya no será accesible en el diccionario. Puedes evitar esto basando los cálculos del código hash en campos inmutables.

A continuación, proporcionaremos un ejemplo completo que ilustra cómo anular GetHashCode.

Anular Equals

Los axiomas para object.Equals son los siguientes:

- Un objeto no puede ser igual a null (a menos que sea un tipo nullable).
- La igualdad es reflexiva (un objeto es igual a sí mismo).
- La igualdad es conmutativa (si a.Equals(b), entonces b.Equals(a)).
- La igualdad es transitiva (si a.Equals(b) y b.Equals(c), entonces a.Equals(c)).
- Las operaciones de igualdad son repetibles y confiables (no generan excepciones).

Sobrecargar == y !=

Además de anular Equals, puedes opcionalmente sobrecargar los operadores de igualdad e desigualdad. Esto se hace casi siempre con estructuras (structs), ya que la consecuencia de no hacerlo es que los operadores == y != simplemente no funcionarán en tu tipo.

Con las clases, hay dos formas de proceder:

- Dejar los operadores == y != como están, para que apliquen igualdad referencial.
- Sobrecargar los operadores == y != en línea con Equals.

El primer enfoque es el más común con tipos personalizados, especialmente con tipos mutables. Asegura que tu tipo cumpla con la expectativa de que los operadores == y != exhiban igualdad referencial con los tipos de referencia, lo que evita confundir a los consumidores. Vimos un ejemplo anteriormente:

```
var sb1 = new StringBuilder ("foo");
var sb2 = new StringBuilder ("foo");
Console.WriteLine (sb1 == sb2);          // False (referential equality)
Console.WriteLine (sb1.Equals (sb2));    // True  (value equality)
```

El segundo enfoque tiene sentido con tipos para los cuales un consumidor nunca desearía igualdad referencial. Estos tipos suelen ser inmutables, como las clases string y System.Uri, y a veces son buenos candidatos para ser structs.



Aunque es posible sobrecargar != para que signifique algo diferente a !(==), esto casi nunca se hace en la práctica, excepto en casos como la comparación de float.NaN.

Implementando IEquatable<T>

Para completitud, es bueno implementar IEquatable<T> al sobrescribir Equals. Los resultados de IEquatable<T> siempre deben coincidir con los resultados del método Equals sobrescrito. Implementar IEquatable<T> no requiere programación adicional si estructuras la implementación del método Equals como se muestra en el siguiente ejemplo.

Un ejemplo: la estructura "Area"

Imagina que necesitamos una estructura para representar un área cuyo ancho y alto son intercambiables. En otras palabras, 5 × 10 es igual a 10 × 5. (Este tipo de estructura sería útil en un algoritmo que organiza formas rectangulares).

Aquí está el código completo:

```
public struct Area : IEquatable <Area>
{
  public readonly int Measure1;
  public readonly int Measure2;

  public Area (int m1, int m2)
  {
    Measure1 = Math.Min (m1, m2);
    Measure2 = Math.Max (m1, m2);
  }

  public override bool Equals (object other)
      => other is Area a && Equals (a);    // Calls method below

  public bool Equals (Area other)        // Implements IEquatable<Area>
    => Measure1 == other.Measure1 && Measure2 == other.Measure2;

  public override int GetHashCode()
    => HashCode.Combine (Measure1, Measure2);

  public static bool operator == (Area a1, Area a2) => a1.Equals (a2);

  public static bool operator != (Area a1, Area a2) => !a1.Equals (a2);
}
```



Desde C# 10, puedes simplificar el proceso utilizando los "records". Al declararlo como un "record struct", puedes eliminar todo el código que sigue al constructor.

Al implementar GetHashCode, utilizamos la función HashCode.Combine de .NET para generar un hash code compuesto. (Antes de que existiera esta función, un enfoque popular era multiplicar cada valor por un número primo y luego sumarlos).

Aquí tienes una demostración de la estructura "Area":

```
Area a1 = new Area (5, 10);
Area a2 = new Area (10, 5);

Console.WriteLine (a1.Equals (a2));    // True
Console.WriteLine (a1 == a2);          // True
```

Comparadores de igualdad intercambiables

Si deseas que un tipo adopte diferentes semánticas de igualdad solo para un escenario específico, puedes usar un IEqualityComparer intercambiable. Esto es particularmente útil en conjunción con las clases de colección estándar, y se describe en el siguiente capítulo, en "Conexión de Igualdad y Orden" en la página 388.

Comparación de orden

Además de definir protocolos estándar para la igualdad, C# y .NET definen dos protocolos estándar para determinar el orden de un objeto en relación con otro:

- Las interfaces IComparable (IComparable e IComparable<T>)
- Los operadores > y <

Las interfaces IComparable se utilizan en algoritmos de clasificación de propósito general. En el siguiente ejemplo, el método estático Array.Sort funciona porque System.String implementa las interfaces IComparable:

```
string[] colors = { "Green", "Red", "Blue" };
Array.Sort (colors);
foreach (string c in colors) Console.Write (c + " ");   // Blue Green Red
```

Los operadores < y > son más especializados y están destinados principalmente a tipos numéricos. Debido a que se resuelven estáticamente, pueden traducirse a bytecode altamente eficiente, adecuado para algoritmos computacionalmente intensivos.

.NET también proporciona protocolos de ordenación intercambiables a través de las interfaces IComparer. Se describen en la sección final del Capítulo 7.

IComparable

Las interfaces IComparable se definen de la siguiente manera:

```
public interface IComparable       { int CompareTo (object other); }
public interface IComparable<in T> { int CompareTo (T other);      }
```

Las dos interfaces representan la misma funcionalidad. Con tipos de valor, la interfaz genérica type-safe es más rápida que la interfaz no genérica. En ambos casos, el método CompareTo funciona de la siguiente manera:

- Si 'a' viene después de 'b', a.CompareTo(b) devuelve un número positivo.
- Si 'a' es igual a 'b', a.CompareTo(b) devuelve 0.
- Si 'a' viene antes de 'b', a.CompareTo(b) devuelve un número negativo.

```
Console.WriteLine ("Beck".CompareTo ("Anne"));      // 1
Console.WriteLine ("Beck".CompareTo ("Beck"));      // 0
Console.WriteLine ("Beck".CompareTo ("Chris"));     // -1
```

La mayoría de los tipos base implementan ambas interfaces IComparable. Estas interfaces a veces también se implementan al escribir tipos personalizados. A continuación, proporcionaremos un ejemplo.

IComparable versus Equals

Considera un tipo que tanto anula Equals como implementa las interfaces IComparable. Esperarías que cuando Equals devuelva true, CompareTo debería devolver 0. Y estarías en lo correcto. Pero aquí está el detalle:

> Cuando Equals devuelve false, CompareTo puede devolver lo que desee (siempre y cuando sea internamente coherente).

En otras palabras, la igualdad puede ser "más estricta" que la comparación, pero no al revés (violentar esto hará que los algoritmos de ordenación se rompan). Así que CompareTo puede decir: "Todos los objetos son iguales", mientras que Equals dice: "¡Pero algunos son más iguales que otros!"

Un gran ejemplo de esto es System.String. El método Equals y el operador == de String utilizan una comparación ordinal, que compara los valores de punto Unicode de cada carácter. Sin embargo, su método CompareTo utiliza una comparación dependiente de la cultura y menos estricta.

En la mayoría de las computadoras, por ejemplo, las cadenas "ü" y "ū" son diferentes según Equals, pero son iguales según CompareTo.

En el Capítulo 7, discutimos el protocolo de ordenación intercambiable, IComparer, que te permite especificar un algoritmo de ordenación alternativo al ordenar o instanciar una colección ordenada. Un IComparer personalizado puede ampliar aún más la diferencia entre CompareTo y Equals; por ejemplo, un comparador de cadenas que no distingue entre mayúsculas y minúsculas devolverá 0 al comparar "A" y "a". Sin embargo, la regla inversa sigue aplicándose: CompareTo nunca puede ser más estricto que Equals.

Cuando implementes las interfaces IComparable en un tipo personalizado, puedes evitar violar esta regla escribiendo la primera línea de CompareTo de la siguiente manera:

```
if (Equals (other)) return 0;
```

Después de eso, puede devolver lo que desee, siempre y cuando sea coherente.

< and >

Algunos tipos definen los operadores < and >. Por ejemplo:

```
bool after2010 = DateTime.Now > new DateTime (2010, 1, 1);
```

Cuando se implementan, se espera que los operadores < and > sean funcionalmente consistentes con las interfaces IComparable. Esta es una práctica estándar en .NET.

También es una práctica estándar implementar las interfaces IComparable siempre que se sobrecarguen < and >, aunque no se cumple al revés. De hecho, la mayoría de los tipos .NET que implementan IComparable no sobrecargan los operadores < y >. Esto difiere de la situación con la igualdad, para la cual es normal sobrecargar == al anular Equals.

Por lo general, los operadores > y < se sobrecargan solo cuando:

- Un tipo tiene un fuerte concepto intrínseco de "mayor que" y "menor que" (en contraste con los conceptos más amplios de "viene antes" y "viene después" de IComparable).
- Solo hay una forma o contexto en el que realizar la comparación.
- El resultado es invariante en todas las culturas.

System.String no cumple con el último punto: los resultados de las comparaciones de cadenas pueden variar según el idioma. Por lo tanto, string no admite los operadores > y <:

```
bool error = "Beck" > "Anne";       // Compile-time error
```

Implementando las interfaces IComparable

En la siguiente estructura que representa una nota musical, implementamos las interfaces IComparable, sobrecargamos los operadores < y >, y también anulamos Equals/GetHashCode y sobrecargamos == y != para completar el ejemplo:

```
public struct Note : IComparable<Note>, IEquatable<Note>, IComparable
{
  int _semitonesFromA;
  public int SemitonesFromA { get { return _semitonesFromA; } }

  public Note (int semitonesFromA)
  {
    _semitonesFromA = semitonesFromA;
  }

  public int CompareTo (Note other)            // Generic IComparable<T>
  {
    if (Equals (other)) return 0;    // Fail-safe check
    return _semitonesFromA.CompareTo (other._semitonesFromA);
  }

  int IComparable.CompareTo (object other)     // Nongeneric IComparable
  {
    if (!(other is Note))
      throw new InvalidOperationException ("CompareTo: Not a note");
    return CompareTo ((Note) other);
  }

  public static bool operator < (Note n1, Note n2)
     => n1.CompareTo (n2) < 0;

  public static bool operator > (Note n1, Note n2)
    => n1.CompareTo (n2) > 0;

  public bool Equals (Note other)    // for IEquatable<Note>
    => _semitonesFromA == other._semitonesFromA;


   public override bool Equals (object other)
   {
     if (!(other is Note)) return false;
     return Equals ((Note) other);
   }

   public override int GetHashCode() => _semitonesFromA.GetHashCode();

   public static bool operator == (Note n1, Note n2) => n1.Equals (n2);

   public static bool operator != (Note n1, Note n2) => !(n1 == n2);
}
```

Las clases de utilidad

Console

La clase estática Console se encarga de la entrada/salida estándar para aplicaciones basadas en la consola. En una aplicación de línea de comandos (consola), la entrada proviene del teclado a través de Read, ReadKey y ReadLine, y la salida se envía a la ventana de texto a través de Write y WriteLine. Puedes controlar la posición y dimensiones de la ventana con las propiedades WindowLeft, WindowTop, WindowHeight y WindowWidth. También puedes cambiar las propiedades BackgroundColor y ForegroundColor y manipular el cursor con las propiedades CursorLeft, CursorTop y CursorSize:

```
Console.WindowWidth = Console.LargestWindowWidth;
Console.ForegroundColor = ConsoleColor.Green;
Console.Write ("test... 50%");
Console.CursorLeft -= 3;
Console.Write ("90%");     // test... 90%
```

Los métodos Write y WriteLine están sobrecargados para aceptar una cadena de formato compuesta (ver String.Format en "Manipulación de cadenas y texto" en la página 271). Sin embargo, ninguno de los métodos acepta un proveedor de formato, por lo que estás limitado a CultureInfo.CurrentCulture. (La solución alternativa, por supuesto, es llamar explícitamente a string.Format).

La propiedad Console.Out devuelve un TextWriter. Pasar Console.Out a un método que espera un TextWriter es una forma útil de hacer que ese método escriba en la consola con fines de diagnóstico.

También puedes redirigir las corrientes de entrada y salida de la consola mediante los métodos SetIn y SetOut:

```
// First save existing output writer:
System.IO.TextWriter oldOut = Console.Out;

// Redirect the console's output to a file:
using (System.IO.TextWriter w = System.IO.File.CreateText
                                ("e:\\output.txt"))
{
  Console.SetOut (w);

  Console.WriteLine ("Hello world");
}

// Restore standard console output
Console.SetOut (oldOut);
```

En el Capítulo 15, describimos cómo funcionan las corrientes (streams) y los escritores de texto (text writers).



Cuando ejecutas aplicaciones WPF o Windows Forms en Visual Studio, la salida de la consola se redirige automáticamente a la ventana de salida de Visual Studio (en modo de depuración). Esto puede hacer que Console.Write sea útil para fines de diagnóstico; aunque en la mayoría de los casos, las clases Debug y Trace en el espacio de nombres System.Diagnostics son más apropiadas (ver Capítulo 13).

Environment

La clase estática System.Environment proporciona una serie de propiedades útiles:

Archivos y carpetas

CurrentDirectory, SystemDirectory, CommandLine

Computadora y sistema operativo

MachineName, ProcessorCount, OSVersion, NewLine

Inicio de sesión de usuario

UserName, UserInteractive, UserDomainName

Diagnóstico

TickCount, StackTrace, WorkingSet, Version

Puedes obtener carpetas adicionales llamando a GetFolderPath; esto lo describimos en "Operaciones de archivo y directorio" en la página 699.

Puedes acceder a las variables de entorno del sistema operativo (lo que ves cuando escribes "set" en el símbolo del sistema) con los siguientes tres métodos: GetEnvironmentVariable, GetEnvironmentVariables y SetEnvironmentVariable.

La propiedad ExitCode te permite establecer el código de retorno, por ejemplo, cuando tu programa es llamado desde un archivo de comandos o de procesamiento por lotes, y el método FailFast termina un programa inmediatamente, sin realizar limpieza.

La clase Environment disponible para aplicaciones de Windows Store ofrece solo un número limitado de miembros (ProcessorCount, NewLine y FailFast).

Process

La clase Process en System.Diagnostics te permite iniciar un nuevo proceso (en el Capítulo 13, describimos cómo también puedes usarla para interactuar con otros procesos en ejecución en la computadora).



Por razones de seguridad, la clase Process no está disponible para aplicaciones de Windows Store, y no puedes iniciar procesos arbitrarios. En su lugar, debes usar la clase Windows.System.Launcher para "lanzar" una URI o un archivo al cual tienes acceso. Por ejemplo:

```
Launcher.LaunchUriAsync (new Uri ("http://albahari.com"));

var file = await KnownFolders.DocumentsLibrary
                             .GetFileAsync ("foo.txt");
Launcher.LaunchFileAsync (file);
```

Esto abre la URI o el archivo utilizando el programa asociado con el esquema de URI o la extensión de archivo. Tu programa debe estar en primer plano para que esto funcione.

El método estático Process.Start tiene varias sobrecargas; la más simple acepta solo el nombre del archivo con argumentos opcionales:

```
Process.Start ("notepad.exe");
Process.Start ("notepad.exe", "e:\\file.txt");
```

La sobrecarga más flexible acepta una instancia de ProcessStartInfo. Con esto, puedes capturar y redirigir la entrada, salida y salida de error del proceso lanzado (si dejas UseShellExecute como false). El siguiente ejemplo captura la salida de llamar a ipconfig:

```
ProcessStartInfo psi = new ProcessStartInfo
{
  FileName = "cmd.exe",
  Arguments = "/c ipconfig /all",
  RedirectStandardOutput = true,
  UseShellExecute = false
};
Process p = Process.Start (psi);
string result = p.StandardOutput.ReadToEnd();
Console.WriteLine (result);
```

Si no rediriges la salida, Process.Start ejecuta el programa en paralelo al llamador. Si deseas esperar a que el nuevo proceso se complete, puedes llamar a WaitForExit en el objeto Process, con un tiempo de espera opcional.

Redireccionar las salidas de error y de los flujos

Con UseShellExecute en false (el valor predeterminado en .NET), puedes capturar las entradas, salidas y salidas de error estándar y luego escribir/leer estos flujos a través de las propiedades StandardInput, StandardOutput y StandardError.

Surge una dificultad cuando necesitas redirigir tanto la salida estándar como los flujos de error estándar, ya que generalmente no puedes saber en qué orden leer los datos de cada uno (porque no sabes de antemano cómo se entrelazarán los datos). La solución es leer de ambos flujos al mismo tiempo, lo cual puedes lograr leyendo al menos uno de los flujos de manera asíncrona. Aquí te muestro cómo hacerlo:

- Manejar los eventos OutputDataReceived y/o ErrorDataReceived. Estos eventos se activan cuando se recibe datos de salida/error.
- Llamar a BeginOutputReadLine y/o BeginErrorReadLine. Esto habilita los eventos mencionados anteriormente.

El siguiente método ejecuta un ejecutable mientras captura tanto la salida como los errores del flujo:

```
(string output, string errors) Run (string exePath, string args = "")
{
  using var p = Process.Start (new ProcessStartInfo (exePath, args)
  {
    RedirectStandardOutput = true,
    RedirectStandardError = true,
    UseShellExecute = false,
  });

  var errors = new StringBuilder ();

  // Read from the error stream asynchronously...
  p.ErrorDataReceived += (sender, errorArgs) =>
  {
    if (errorArgs.Data != null) errors.AppendLine (errorArgs.Data);
  };
  p.BeginErrorReadLine ();

  // ...while we read from the output stream synchronously:
  string output = p.StandardOutput.ReadToEnd();

  p.WaitForExit();
  return (output, errors.ToString());
}
```

UseShellExecute



En .NET 5+ (y .NET Core), el valor predeterminado para UseShellExecute es false, mientras que en .NET Framework era true. Debido a que esto es un cambio que puede causar problemas, es recomendable revisar todas las llamadas a Process.Start al portar código desde .NET Framework.

La bandera UseShellExecute cambia cómo el CLR inicia el proceso. Con UseShellExecute en true, puedes hacer lo siguiente:

- Especificar una ruta a un archivo o documento en lugar de un ejecutable (lo que resulta en que el sistema operativo abra el archivo o documento con su aplicación asociada)
- Especificar una URL (lo que resulta en que el sistema operativo navegue a esa URL en el navegador web predeterminado)
- (Solo en Windows) Especificar un verbo (como "runas", para ejecutar el proceso con elevación administrativa).

La desventaja es que no puedes redirigir las entradas o salidas de los flujos. Si necesitas hacerlo al ejecutar un archivo o documento, una solución alternativa es establecer UseShellExecute en false e invocar el proceso de línea de comandos (cmd.exe) con el interruptor "/c", como hicimos anteriormente al llamar a ipconfig.

En Windows, UseShellExecute indica al CLR que utilice la función Windows ShellExecute en lugar de la función CreateProcess. En Linux, UseShellExecute indica al CLR que llame a xdg-open, gnome-open o kfmclient.

AppContext

La clase estática System.AppContext expone dos propiedades útiles:

- BaseDirectory devuelve la carpeta en la que se inició la aplicación. Esta carpeta es importante para la resolución de ensamblados (encontrar y cargar dependencias) y para localizar archivos de configuración (como appsettings.json).
- TargetFrameworkName te indica el nombre y la versión del tiempo de ejecución .NET al que la aplicación se dirige (como se especifica en su archivo .runtimeconfig.json). Esto puede ser más antiguo que el tiempo de ejecución que realmente se está utilizando.

Además, la clase AppContext administra un diccionario global de valores booleanos con clave de cadena, destinado a ofrecer a los escritores de bibliotecas un mecanismo estándar para permitir a los consumidores activar o desactivar nuevas características. Este enfoque sin tipo tiene sentido con características experimentales que deseas mantener sin documentar para la mayoría de los usuarios.

El consumidor de una biblioteca solicita que habilites una característica de la siguiente manera:

```
AppContext.SetSwitch ("MyLibrary.SomeBreakingChange", true);
```

El código dentro de esa biblioteca puede verificar ese interruptor de la siguiente manera:

```
bool isDefined, switchValue;
isDefined = AppContext.TryGetSwitch ("MyLibrary.SomeBreakingChange",
                                     out switchValue);
```

TryGetSwitch devuelve false si el interruptor no está definido. Esto te permite distinguir un interruptor indefinido de aquellos cuyo valor está establecido en false, si es necesario.



Irónicamente, el diseño de TryGetSwitch ilustra cómo no escribir API. El parámetro de salida no es necesario y el método debería devolver en su lugar un bool anulable cuyo valor sea true, false o null para indefinido. Esto permitiría el siguiente uso:

```
bool switchValue = AppContext.GetSwitch ("...") ?? false;
```

7 Colecciones

.NET proporciona un conjunto estándar de tipos para almacenar y gestionar colecciones de objetos. Estos incluyen listas redimensionables, listas enlazadas, diccionarios ordenados y no ordenados, y arrays. De estos, solo los arrays forman parte del lenguaje C#; las demás colecciones son solo clases que se instancian como cualquier otra.

Podemos dividir los tipos en el BCL de .NET para colecciones en las siguientes categorías:

- Interfaces que definen protocolos estándar de colección.
- Clases de colección listas para usar (listas, diccionarios, etc.).
- Clases base para escribir colecciones específicas de la aplicación.

Este capítulo cubre cada una de estas categorías, con una sección adicional sobre los tipos utilizados para determinar la igualdad y el orden de los elementos.

Los espacios de nombres de las colecciones son los siguientes:

Namespace	Contains
`System.Collections`	Nongeneric collection classes and interfaces
`System.Collections.Specialized`	Strongly typed nongeneric collection classes
`System.Collections.Generic`	Generic collection classes and interfaces
`System.Collections.ObjectModel`	Proxies and bases for custom collections
`System.Collections.Concurrent`	Thread-safe collections (see Chapter 22)

Enumeración

En informática, existen muchos tipos diferentes de colecciones, que van desde estructuras de datos simples, como arrays o listas enlazadas, hasta otras más complejas, como árboles rojo/negro y tablas hash. Aunque la implementación interna y las características externas de estas estructuras de datos varían ampliamente, la capacidad de recorrer el contenido de la colección es una necesidad casi universal. El BCL de .NET satisface esta necesidad a través de un par de interfaces (IEnumerable e IEnumerator, y sus contrapartes genéricas) que permiten que diferentes estructuras de datos expongan una API de recorrido común. Estas forman parte de un conjunto más amplio de interfaces de colección ilustradas en la Figura 7-1.

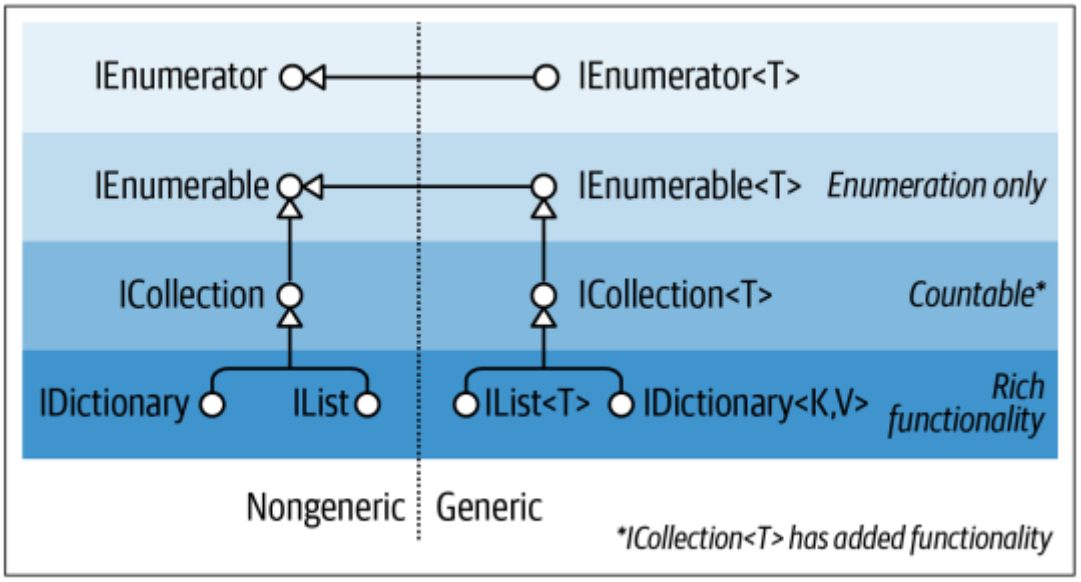


Figure 7-1. Collection interfaces

IEnumerable y IEnumerator

La interfaz IEnumerator define el protocolo básico de bajo nivel mediante el cual los elementos de una colección se recorren o enumeran de forma unidireccional. Su declaración es la siguiente:

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

El método MoveNext avanza el elemento actual o "cursor" a la siguiente posición y devuelve false si no hay más elementos en la colección. El método Current devuelve el elemento en la posición actual (normalmente convertido de tipo object a un tipo más específico). Se debe llamar a MoveNext antes de recuperar el primer elemento, esto se hace para permitir una colección vacía. El método Reset, si está implementado, vuelve al inicio, lo que permite enumerar la colección nuevamente. Reset existe principalmente para la interoperabilidad con Component Object Model (COM); generalmente se evita llamarlo directamente debido a que no es universalmente compatible y no es necesario, ya que generalmente es igual de fácil instanciar un nuevo enumerador.

Las colecciones generalmente no implementan enumeradores; en su lugar, proporcionan enumeradores a través de la interfaz IEnumerable:

```
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

Al definir un único método que devuelve un enumerador, IEnumerable proporciona flexibilidad en el sentido de que la lógica de iteración puede ser delegada a otra clase. Además, esto significa que varios consumidores pueden enumerar la colección al mismo tiempo sin interferir entre sí. Se puede pensar en IEnumerable como "IEnumeratorProvider", y es la interfaz más básica que implementan las clases de colección.

El siguiente ejemplo ilustra el uso a nivel de bajo nivel de IEnumerable e IEnumerator:

```
string s = "Hello";

// Because string implements IEnumerable, we can call GetEnumerator():
IEnumerator rator = s.GetEnumerator();

while (rator.MoveNext())
{
  char c = (char) rator.Current;
  Console.Write (c + ".");
}

// Output:  H.e.l.l.o.
```

Sin embargo, es raro llamar a los métodos en los enumeradores directamente de esta manera, porque C# proporciona un atajo sintáctico: la instrucción foreach. Aquí está el mismo ejemplo reescrito usando foreach:

```
string s = "Hello";      // The String class implements IEnumerable

foreach (char c in s)
  Console.Write (c + ".");
```

IEnumerable<T> y IEnumerator<T>

IEnumerator e IEnumerable casi siempre se implementan en conjunto con sus versiones genéricas extendidas:

```
public interface IEnumerator<T> : IEnumerator, IDisposable
{
  T Current { get; }
}

public interface IEnumerable<T> : IEnumerable
{
  IEnumerator<T> GetEnumerator();
}
```

Al definir una versión tipada de Current y GetEnumerator, estas interfaces fortalecen la seguridad de tipos estáticos, evitan la sobrecarga de boxing con elementos de tipo de valor y son más convenientes para el consumidor. Los arrays implementan automáticamente IEnumerable<T> (donde T es el tipo de miembro del array).

Gracias a la mejora en la seguridad de tipos estáticos, llamar al siguiente método con un array de caracteres generará un error en tiempo de compilación:

```
void Test (IEnumerable<int> numbers) { ... }
```

Es una práctica estándar para las clases de colección exponer públicamente IEnumerable<T> mientras "ocultan" IEnumerable no genérico a través de la implementación explícita de la interfaz. Esto se hace para que si llama directamente a GetEnumerator(), obtenga el IEnumerator<T> genérico seguro de tipos. Sin embargo, hay momentos en los que esta regla se rompe por razones de compatibilidad hacia atrás (los genéricos no existían antes de C# 2.0). Un

buen ejemplo son los arrays, que deben devolver el IEnumerator no genérico (la forma elegante de decirlo es "clásico") para evitar romper el código anterior. Para obtener un IEnumerator<T> genérico, debes realizar una conversión para exponer la interfaz explícita:

```
int[] data = { 1, 2, 3 };
var rator = ((IEnumerable <int>)data).GetEnumerator();
```

Afortunadamente, rara vez necesitas escribir este tipo de código gracias a la instrucción foreach.

IEnumerable<T> e IDisposable

IEnumerator<T> hereda de IDisposable. Esto permite que los enumeradores mantengan referencias a recursos como conexiones de bases de datos y aseguren que esos recursos se liberen cuando la enumeración se completa (o se abandona a mitad de camino). La instrucción foreach reconoce este detalle y traduce lo siguiente:

```
foreach (var element in somethingEnumerable) { ... }
```

en el equivalente lógico de esto:

```
using (var rator = somethingEnumerable.GetEnumerator())
  while (rator.MoveNext())
  {
    var element = rator.Current;
    ...
  }
```

¿Cuándo usar las interfaces no genéricas?

Dado el mayor nivel de seguridad de tipos de las interfaces genéricas de colección, como IEnumerable<T>, surge la pregunta: ¿alguna vez necesitas usar IEnumerable no genérico (o ICollection o IList)?

En el caso de IEnumerable, debes implementar esta interfaz junto con IEnumerable<T>, ya que la última deriva de la primera. Sin embargo, es muy raro que realmente implementes estas interfaces desde cero: en casi todos los casos, puedes adoptar el enfoque de nivel superior utilizando métodos iteradores, Collection<T> y LINQ. Entonces, ¿qué hay de los consumidores? En casi todos los casos, puedes administrar completamente con las interfaces genéricas. Sin embargo, las interfaces no genéricas aún son ocasionalmente útiles debido a su capacidad para proporcionar unificación de tipos para colecciones de todos los tipos de elementos. El siguiente método, por ejemplo, cuenta elementos en cualquier colección de forma recursiva:

```
public static int Count (IEnumerable e)
{
  int count = 0;
  foreach (object element in e)
  {
    var subCollection = element as IEnumerable;
    if (subCollection != null)
      count += Count (subCollection);
    else
      count++;
  }
  return count;
}
```

Aunque C# ofrece covarianza con las interfaces genéricas, podría parecer válido hacer que este método acepte IEnumerable<object>. Sin embargo, esto fallaría con elementos de tipo valor y con colecciones heredadas que no implementan IEnumerable<T>, como ControlCollection en Windows Forms.

(En un tema ligeramente tangente, es posible que hayas notado un error potencial en nuestro ejemplo: las referencias cíclicas causarán una recursión infinita y harán que el método falle. Podríamos solucionar esto más fácilmente utilizando un HashSet (consulta "HashSet<T> and SortedSet<T>" en la página 370).

El bloque "using" garantiza la eliminación adecuada de recursos. Obtendrás más información sobre IDisposable en el Capítulo 12.

Implementar las interfaces de enumeración

Es posible que desees implementar IEnumerable o IEnumerable<T> por una o más de las siguientes razones:

- Para admitir la declaración foreach.
- Para interoperar con cualquier cosa que espere una colección estándar.
- Para cumplir con los requisitos de una interfaz de colección más sofisticada.
- Para admitir inicializadores de colección.

Para implementar IEnumerable/IEnumerable<T>, debes proporcionar un enumerador. Puedes hacerlo de una de las siguientes tres formas:

- Si la clase está "envolviendo" otra colección, devolviendo el enumerador de la colección envuelta.
- A través de un iterador usando la instrucción yield return.
- Mediante la creación de tu propia implementación de IEnumerator/IEnumerator<T>.



También puedes heredar de una colección existente: Collection<T> está diseñada específicamente para este propósito (consulte "Colecciones personalizables y proxies" en la página 378). Otra opción es utilizar los operadores de consulta LINQ, que se cubren en el Capítulo 8.

Devolver el enumerador de otra colección es simplemente cuestión de llamar a GetEnumerator en la colección interna. Sin embargo, esto solo es viable en los escenarios más simples en los que los elementos de la colección interna son exactamente lo que se requiere. Un enfoque más flexible es escribir un iterador utilizando la instrucción yield return de C#. Un iterador es una característica del lenguaje C# que ayuda a escribir colecciones, de la misma manera en que la declaración foreach ayuda a consumir colecciones. Un iterador maneja automáticamente la implementación de IEnumerable e IEnumerator, o sus versiones genéricas. Aquí tienes un ejemplo sencillo:

```
public class MyCollection : IEnumerable
{
  int[] data = { 1, 2, 3 };

  public IEnumerator GetEnumerator()
  {
    foreach (int i in data)
      yield return i;
  }
}
```

Observa la "magia negra": ¡GetEnumerator no parece devolver un enumerador en absoluto! Al analizar la instrucción yield return, el compilador genera una clase enumeradora anidada oculta y luego refactoriza GetEnumerator para instanciar y devolver esa clase. Los iteradores son potentes y sencillos (y se utilizan ampliamente en la implementación de los operadores de consulta estándar de LINQ-to-Object).

Siguiendo este enfoque, también podemos implementar la interfaz genérica IEnumerable<T>:

```
public class MyGenCollection : IEnumerable<int>
{
  int[] data = { 1, 2, 3 };

  public IEnumerator<int> GetEnumerator()
  {
    foreach (int i in data)
      yield return i;
  }

  // Explicit implementation keeps it hidden:
  IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Dado que IEnumerable<T> hereda de IEnumerable, debemos implementar ambas versiones, la genérica y la no genérica, de GetEnumerator. Siguiendo la práctica estándar, hemos implementado explícitamente la versión no genérica. Puede simplemente llamar a GetEnumerator genérico porque IEnumerator<T> hereda de IEnumerator.

La clase que acabamos de escribir sería adecuada como base para escribir una colección más sofisticada. Sin embargo, si no necesitamos más que una implementación simple de IEnumerable<T>, la instrucción yield return permite una variación más fácil. En lugar de escribir una clase, puedes mover la lógica de iteración a un método que devuelve un IEnumerable<T> genérico y dejar que el compilador se encargue del resto. Aquí tienes un ejemplo:

```
public static IEnumerable <int> GetSomeIntegers()
{
  yield return 1;
  yield return 2;
  yield return 3;
}
```

Aquí tienes cómo se utiliza nuestro método:

```
foreach (int i in Test.GetSomeIntegers())
  Console.WriteLine (i);
```

El enfoque final para escribir GetEnumerator es escribir una clase que implemente IEnumerator directamente. Esto es exactamente lo que hace el compilador detrás de escenas al resolver iteradores. (Afortunadamente, rara vez necesitarás llegar tan lejos tú mismo). El siguiente ejemplo define una colección que contiene de forma rígida los enteros 1, 2 y 3:

```
public class MyIntList : IEnumerable
{
  int[] data = { 1, 2, 3 };

  public IEnumerator GetEnumerator() => new Enumerator (this);

  class Enumerator : IEnumerator       // Define an inner class
  {                                    // for the enumerator.
    MyIntList collection;
    int currentIndex = -1;

    public Enumerator (MyIntList items) => this.collection = items;

    public object Current
    {
      get
      {
        if (currentIndex == -1)
          throw new InvalidOperationException ("Enumeration not started!");
        if (currentIndex == collection.data.Length)
          throw new InvalidOperationException ("Past end of list!");
        return collection.data [currentIndex];
      }
    }

    public bool MoveNext()

    {
      if (currentIndex >= collection.data.Length - 1) return false;
      return ++currentIndex < collection.data.Length;
    }

    public void Reset() => currentIndex = -1;
  }
}
```



Implementar Reset es opcional; en su lugar, puedes lanzar una NotSupportedException.

Ten en cuenta que la primera llamada a MoveNext debería moverse al primer elemento (y no al segundo) de la lista.

Para estar a la par con un iterador en funcionalidad, también debemos implementar IEnumerator<T>. Aquí tienes un ejemplo omitiendo la comprobación de límites por brevedad:

```
class MyIntList : IEnumerable<int>
{
  int[] data = { 1, 2, 3 };

  // The generic enumerator is compatible with both IEnumerable and
  // IEnumerable<T>. We implement the nongeneric GetEnumerator method
  // explicitly to avoid a naming conflict.

  public IEnumerator<int> GetEnumerator() => new Enumerator(this);
  IEnumerator IEnumerable.GetEnumerator() => new Enumerator(this);

  class Enumerator : IEnumerator<int>
  {
    int currentIndex = -1;
    MyIntList collection;

    public Enumerator (MyIntList items) => this.items = items;

    public int Current => collection.data [currentIndex];
    object IEnumerator.Current => Current;

    public bool MoveNext() => ++currentIndex < collection.data.Length;

    public void Reset() => currentIndex = -1;

    // Given we don't need a Dispose method, it's good practice to
    // implement it explicitly, so it's hidden from the public interface.
    void IDisposable.Dispose() {}
  }
}
```

El ejemplo con genéricos es más rápido porque IEnumerator<int>.Current no requiere la conversión de int a object, evitando así el costo de la boxing.

Las interfaces ICollection e IList

Aunque las interfaces de enumeración proporcionan un protocolo para la iteración de solo avance sobre una colección, no proporcionan un mecanismo para determinar el tamaño de la colección, acceder a un miembro por índice, buscar o modificar la colección. Para obtener esa funcionalidad, .NET define las interfaces ICollection, IList y IDictionary.

Cada una de ellas tiene versiones genéricas y no genéricas; sin embargo, las versiones no genéricas existen principalmente para soporte heredado.

La Figura 7-1 mostró la jerarquía de herencia de estas interfaces. La forma más sencilla de resumirlas es la siguiente:

IEnumerable<T> (y IEnumerable)

Proporciona la funcionalidad mínima (solo enumeración)

ICollection<T> (y ICollection)

Proporciona una funcionalidad intermedia (por ejemplo, la propiedad Count)

IList<T>/IDictionary<K,V> y sus versiones no genéricas

Proporciona la máxima funcionalidad (incluyendo acceso "aleatorio" por índice/clave)


Es raro que necesites implementar alguna de estas interfaces. En casi todos los casos en los que necesites escribir una clase de colección, puedes en su lugar subclasificar Collection<T> (ver "Colecciones personalizables y proxies" en la página 378). LINQ proporciona otra opción que cubre muchos escenarios.

Las versiones genéricas y no genéricas difieren en formas que podrías no esperar, especialmente en el caso de ICollection. Las razones de esto son principalmente históricas: debido a que las genéricas se desarrollaron posteriormente, las interfaces genéricas se diseñaron con el beneficio de la experiencia, lo que condujo a una elección diferente (y mejor) de los miembros.

Por esta razón, ICollection<T> no extiende ICollection, IList<T> no extiende IList, e IDictionary<TKey, TValue> no extiende IDictionary. Por supuesto, una clase de colección puede implementar ambas versiones de una interfaz si es beneficioso (lo cual suele serlo).


Otra razón más sutil por la que IList<T> no extiende IList es que si lo hiciera, la conversión a IList<T> devolvería una interfaz con los miembros Add(T) y Add(object). Esto efectivamente socavaría la seguridad de tipos estática, ya que podrías llamar a Add con un objeto de cualquier tipo.

Esta sección cubre ICollection<T>, IList<T> y sus versiones no genéricas; "Diccionarios" en la página 372 cubre las interfaces de diccionario.


No hay una justificación coherente en la forma en que las palabras "colección" y "lista" se aplican en las bibliotecas de .NET. Por ejemplo, debido a que IList<T> es una versión más funcional de ICollection<T>, podrías esperar que la clase List<T> sea correspondientemente más funcional que la clase Collection<T>. Pero este no es el caso. Es mejor considerar los términos "colección" y "lista" como ampliamente sinónimos, excepto cuando se trata de un tipo específico.

ICollection<T> e ICollection

ICollection<T> es la interfaz estándar para colecciones de objetos contables. Proporciona la capacidad de determinar el tamaño de una colección (Count), determinar si un elemento existe en la colección (Contains), copiar la colección en un array (ToArray) y determinar si la colección es de solo lectura (IsReadOnly). Para colecciones modificables, también se pueden agregar (Add), eliminar (Remove) y borrar (Clear) elementos de la colección. Y dado que extiende IEnumerable<T>, también se puede recorrer mediante la instrucción foreach:

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
  int Count { get; }

  bool Contains (T item);
  void CopyTo (T[] array, int arrayIndex);
  bool IsReadOnly { get; }

  void Add(T item);
  bool Remove (T item);
  void Clear();
}
```

La interfaz no genérica ICollection es similar en el sentido de que proporciona una colección contable, pero no proporciona funcionalidad para modificar la lista o verificar la pertenencia de elementos:

```
public interface ICollection : IEnumerable
{
   int Count { get; }
   bool IsSynchronized { get; }
   object SyncRoot { get; }
   void CopyTo (Array array, int index);
}
```

La interfaz no genérica también define propiedades para ayudar con la sincronización (Capítulo 14). Estas propiedades se eliminaron en la versión genérica porque la seguridad de subprocesos ya no se considera intrínseca a la colección.

Ambas interfaces son bastante sencillas de implementar. Si se implementa un ICollection<T> de solo lectura, los métodos Add, Remove y Clear deben lanzar una excepción NotSupportedException.

Estas interfaces se suelen implementar en conjunto con la interfaz IList o IDictionary.

IList<T> e IList

IList<T> es la interfaz estándar para colecciones indexables por posición. Además de la funcionalidad heredada de ICollection<T> e IEnumerable<T>, proporciona la capacidad de leer o escribir un elemento por posición (a través de un indexador) e insertar/eliminar por posición:

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
  T this [int index] { get; set; }
  int IndexOf (T item);
  void Insert (int index, T item);
  void RemoveAt (int index);
}
```

Los métodos IndexOf realizan una búsqueda lineal en la lista y devuelven -1 si el elemento especificado no se encuentra.

La versión no genérica de IList tiene más miembros porque hereda menos de ICollection:

```
public interface IList : ICollection, IEnumerable
{
  object this [int index] { get; set }
  bool IsFixedSize { get; }
  bool IsReadOnly  { get; }
  int  Add      (object value);
  void Clear();
  bool Contains (object value);
  int  IndexOf  (object value);
  void Insert   (int index, object value);
  void Remove   (object value);
  void RemoveAt (int index);
}
```

El método Add en la interfaz no genérica IList devuelve un entero: este es el índice del elemento recién agregado. En contraste, el método Add en ICollection<T> tiene un tipo de retorno void.

La clase List<T> de propósito general es la implementación por excelencia tanto de IList<T> como de IList. Los arrays en C# también implementan tanto las versiones genéricas como las no genéricas de IList (aunque los métodos que agregan o eliminan elementos están ocultos mediante implementación de interfaz explícita y lanzarán una NotSupportedException si se llaman).

 Se produce una ArgumentException si intentas acceder a una matriz multidimensional a través del indexador de IList. Esto puede ser problemático al escribir métodos como el siguiente:

```
public object FirstOrNull (IList list)
{
  if (list == null || list.Count == 0) return null;
  return list[0];
}
```

Esto puede parecer infalible, pero lanzará una excepción si se llama con una matriz multidimensional. Puedes verificar si es una matriz multidimensional en tiempo de ejecución con la siguiente expresión (más detalles en el Capítulo 19):

```
list.GetType().IsArray && list.GetType().GetArrayRank()>1
```

IReadOnlyCollection<T> y IReadOnlyList<T>

.NET también define interfaces de colección y lista que exponen solo los miembros necesarios para operaciones de solo lectura:

```
public interface IReadOnlyCollection<out T> : IEnumerable<T>, IEnumerable
{
  int Count { get; }
}

public interface IReadOnlyList<out T> : IReadOnlyCollection<T>,
                                        IEnumerable<T>, IEnumerable
{
  T this[int index] { get; }
}
```

Debido a que el parámetro de tipo en estas interfaces se utiliza solo en posiciones de salida, se marca como covariante. Esto permite tratar una lista de gatos, por ejemplo, como una lista de solo lectura de animales. En contraste, T no se marca como covariante en ICollection<T> e IList<T>, ya que se utiliza tanto en posiciones de entrada como de salida.



Estas interfaces representan una vista de solo lectura de una colección o lista; la implementación subyacente aún puede ser modificable. La mayoría de las colecciones modificables implementan tanto las interfaces de solo lectura como las de lectura/escritura.

Además de permitir trabajar con colecciones de forma covariante, las interfaces de solo lectura permiten que una clase exponga públicamente una vista de solo lectura de una colección modificable privada. Esto se demuestra, junto con una solución mejor, en "ReadOnlyCollection<T>" en la página 383.

IReadOnlyList<T> se mapea al tipo IVectorView<T> en Windows Runtime.

La clase Array

La clase Array es la clase base implícita para todos los arrays unidimensionales y multidimensionales, y es uno de los tipos más fundamentales que implementa las interfaces estándar de colecciones. La clase Array proporciona unificación de tipos, por lo que un conjunto común de métodos está disponible para todos los arrays, independientemente de su declaración o tipo de elemento subyacente.

Debido a que los arrays son tan fundamentales, C# proporciona una sintaxis explícita para su declaración e inicialización, que describimos en los Capítulos 2 y 3. Cuando se declara un array utilizando la sintaxis de C#, el CLR subtipa implícitamente la clase Array, sintetizando un pseudotipo adecuado para las dimensiones y tipos de elemento del array. Este pseudotipo implementa las interfaces de colección genéricas tipadas, como IList<string>.

El CLR también trata a los tipos de arrays de manera especial durante la construcción, asignándoles un espacio contiguo en la memoria. Esto hace que el acceso a los elementos del array sea altamente eficiente, pero evita que se puedan redimensionar más adelante.

Array implementa las interfaces de colección hasta IList<T> tanto en sus formas genéricas como no genéricas. Sin embargo, IList<T> se implementa de forma explícita para mantener la interfaz pública de Array limpia de métodos como Add o Remove, que lanzan una excepción en colecciones de longitud fija como los arrays. La clase Array en realidad ofrece un método estático Resize, aunque esto funciona creando un nuevo array y luego copiando cada elemento. Además de ser ineficiente, las referencias al array en otras partes del programa seguirán apuntando a la versión original. Una solución mejor para colecciones redimensionables es usar la clase List<T> (descrita en la siguiente sección).

Un array puede contener elementos de tipo valor o de tipo referencia. Los elementos de tipo valor se almacenan en el array en su lugar, por lo que un array de tres enteros largos (cada uno de 8 bytes) ocupará 24 bytes de memoria contigua. Sin embargo, un elemento de tipo referencia ocupa solo tanto espacio en el array como una referencia (4 bytes en un entorno de 32 bits o 8 bytes en un entorno de 64 bits). La Figura 7-2 ilustra el efecto en memoria del siguiente programa:

```
StringBuilder[] builders = new StringBuilder [5];
builders [0] = new StringBuilder ("builder1");
builders [1] = new StringBuilder ("builder2");
builders [2] = new StringBuilder ("builder3");

long[] numbers = new long [3];
numbers [0] = 12345;
numbers [1] = 54321;
```

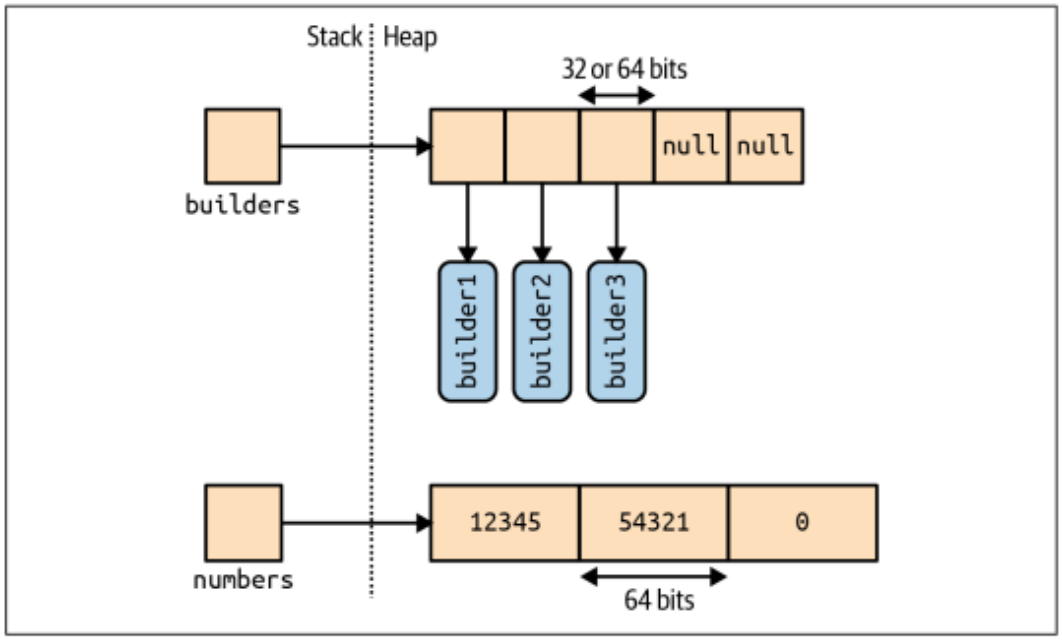


Figure 7-2. Arrays in memory

Debido a que Array es una clase, los arrays siempre son tipos de referencia en sí mismos, independientemente del tipo de elemento del array. Esto significa que la declaración arrayB = arrayA resulta en dos variables que hacen referencia al mismo array. Del mismo modo, dos arrays

distintos siempre fallarán una prueba de igualdad, a menos que se utilice un comparador de igualdad estructural, que compara cada elemento del array:

```
object[] a1 = { "string", 123, true };
object[] a2 = { "string", 123, true };

Console.WriteLine (a1 == a2);                          // False
Console.WriteLine (a1.Equals (a2));                    // False

IStructuralEquatable se1 = a1;
Console.WriteLine (se1.Equals (a2,
 StructuralComparisons.StructuralEqualityComparer));   // True
```

Los arrays se pueden duplicar llamando al método Clone: arrayB = arrayA.Clone(). Sin embargo, esto resulta en un clon superficial, lo que significa que solo se copia la memoria representada por el propio array. Si el array contiene objetos de tipo valor, se copian los valores en sí; si el array contiene objetos de tipo referencia, solo se copian las referencias (lo que resulta en dos arrays cuyos miembros hacen referencia a los mismos objetos).

La Figura 7-3 muestra el efecto de agregar el siguiente código a nuestro ejemplo:

```
StringBuilder[] builders2 = builders;
StringBuilder[] shallowClone = (StringBuilder[]) builders.Clone();
```

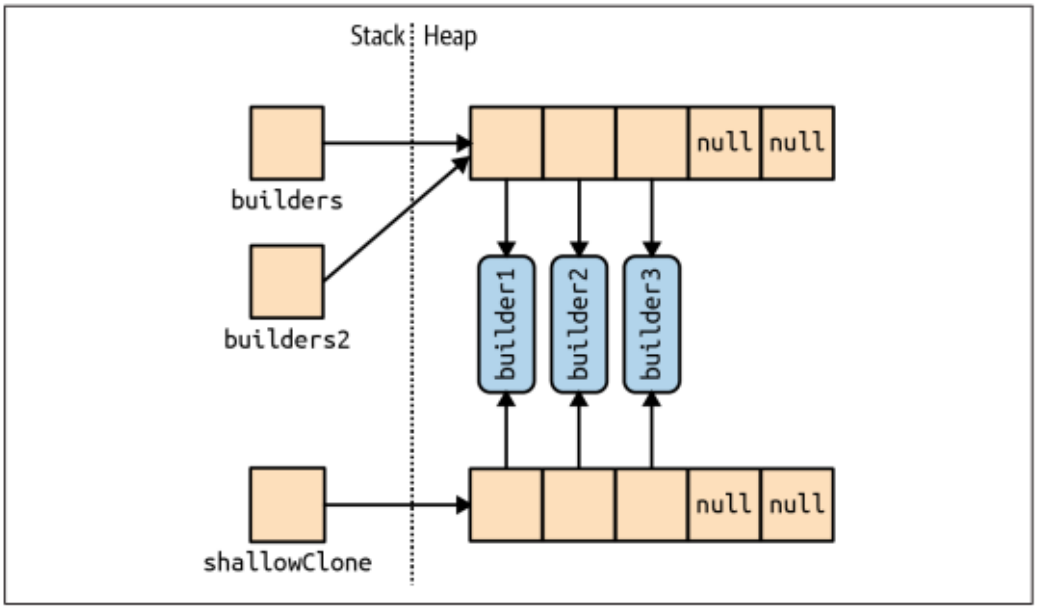


Figure 7-3. Shallow-cloning an array

Para crear una copia profunda, en la cual los subobjetos de tipo referencia se duplican, debes recorrer el array y clonar cada elemento manualmente. Las mismas reglas se aplican a otros tipos de colección en .NET.

Aunque Array está diseñado principalmente para su uso con índices de 32 bits, también tiene soporte limitado para índices de 64 bits (permitiendo teóricamente que un array aborde hasta 264 elementos) a través de varios métodos que aceptan parámetros Int32 e Int64. Estas sobrecargas son inútiles en la práctica porque el CLR no permite que ningún objeto, incluidos los arrays, supere los dos gigabytes de tamaño (ya sea en un entorno de 32 o 64 bits).



Muchos de los métodos de la clase Array que esperarías que sean métodos de instancia son en realidad métodos estáticos. Esta es una decisión de diseño extraña y significa que debes buscar tanto métodos estáticos como de instancia al buscar un método en Array.

Construcción e indexación

La forma más sencilla de crear e indexar arrays es a través de los constructos del lenguaje C#:

```
int[] myArray = { 1, 2, 3 };
int first = myArray [0];
int last = myArray [myArray.Length - 1];
```

Alternativamente, puedes instanciar un array de forma dinámica llamando a Array.CreateInstance. Esto te permite especificar el tipo de elemento y la cantidad de dimensiones en tiempo de ejecución, así como permitir arrays con base no cero al especificar un límite inferior. Las arrays con base no cero no son compatibles con la Especificación del Lenguaje .NET Common Collections (CLS) y no deben ser expuestas como miembros públicos en una biblioteca que pueda ser utilizada por un programa escrito en F# o Visual Basic.

Los métodos GetValue y SetValue te permiten acceder a elementos en un array creado dinámicamente (también funcionan en arrays ordinarios):

```
// Create a string array 2 elements in length:
Array a = Array.CreateInstance (typeof(string), 2);
a.SetValue ("hi", 0);                          //  → a[0] = "hi";
a.SetValue ("there", 1);                       //  → a[1] = "there";
string s = (string) a.GetValue (0);            //  → s = a[0];

// We can also cast to a C# array as follows:
string[] cSharpArray = (string[]) a;
string s2 = cSharpArray [0];
```

Los arrays con índice cero creado dinámicamente se pueden convertir a un array de C# de un tipo coincidente o compatible (según las reglas estándar de varianza de arrays). Por ejemplo, si Apple es una subclase de Fruit, se puede convertir Apple[] a Fruit[]. Esto plantea la cuestión de por qué no se utilizó object[] como tipo de array unificado en lugar de la clase Array. La respuesta es que object[] es incompatible tanto con arrays multidimensionales como con arrays de tipos de valor (y arrays con base no cero). Un array int[] no se puede convertir a object[]. Por lo tanto, necesitamos la clase Array para una unificación de tipos completa.

GetValue y SetValue también funcionan en arrays creados por el compilador, y son útiles al escribir métodos que pueden manejar un array de cualquier tipo y dimensión. Para arrays multidimensionales, aceptan un array de índices:

```
public object GetValue (params int[] indices)
public void   SetValue (object value, params int[] indices)
```

El siguiente método imprime el primer elemento de cualquier array, independientemente de su dimensión:

```
void WriteFirstValue (Array a)
{
  Console.Write (a.Rank + "-dimensional; ");

  // The indexers array will automatically initialize to all zeros, so
  // passing it into GetValue or SetValue will get/set the zero-based
  // (i.e., first) element in the array.

  int[] indexers = new int[a.Rank];
  Console.WriteLine ("First value is " +  a.GetValue (indexers));
}

void Demo()
{
  int[]  oneD = { 1, 2, 3 };
  int[,] twoD = { {5,6}, {8,9} };

  WriteFirstValue (oneD);   // 1-dimensional; first value is 1
  WriteFirstValue (twoD);   // 2-dimensional; first value is 5
}
```



Para trabajar con arrays de tipo desconocido pero con rango conocido, los genéricos proporcionan una solución más fácil y eficiente:

```
void WriteFirstValue<T> (T[] array)
{
  Console.WriteLine (array[0]);
}
```

SetValue lanza una excepción si el elemento es de un tipo incompatible con el array.

Cuando se instancia un array, ya sea a través de la sintaxis del lenguaje o Array.CreateInstance, sus elementos se inicializan automáticamente. Para arrays con elementos de tipo referencia, esto significa asignarles el valor null; para arrays con elementos de tipo valor, esto significa llamar al constructor predeterminado del tipo valor (efectivamente "poner a cero" los miembros). La clase Array también proporciona esta funcionalidad bajo demanda a través del método Clear:

```
public static void Clear (Array array, int index, int length);
```

Este método no afecta el tamaño del array. Esto contrasta con el uso habitual de Clear (como en ICollection<T>.Clear), donde la colección se reduce a cero elementos.

Enumeración

Los arrays se pueden enumerar fácilmente con una declaración foreach:

```
int[] myArray = { 1, 2, 3};
foreach (int val in myArray)
  Console.WriteLine (val);
```

También puedes enumerar usando el método estático Array.ForEach, definido de la siguiente manera:

```
public static void ForEach<T> (T[] array, Action<T> action);
```

Esto utiliza un delegado Action, con esta firma:

```
public delegate void Action<T> (T obj);
```

Aquí está el primer ejemplo reescrito con Array.ForEach:

```
Array.ForEach (new[] { 1, 2, 3 }, Console.WriteLine);
```

Longitud (Length) y Rango (Rank)

La clase Array proporciona los siguientes métodos y propiedades para consultar la longitud y el rango:

```
public int  GetLength      (int dimension);
public long GetLongLength  (int dimension);

public int  Length       { get; }
public long LongLength   { get; }

public int GetLowerBound (int dimension);
public int GetUpperBound (int dimension);

public int Rank { get; }    // Returns number of dimensions in array
```

GetLength y GetLongLength devuelven la longitud de una dimensión dada (0 para un arreglo unidimensional), y Length y LongLength devuelven el número total de elementos en el arreglo, incluyendo todas las dimensiones.

GetLowerBound y GetUpperBound son útiles con arreglos indexados que no comienzan en cero. GetUpperBound devuelve el mismo resultado que agregar GetLowerBound a GetLength para cualquier dimensión dada.

Búsqueda

La clase Array ofrece una variedad de métodos para encontrar elementos dentro de un arreglo unidimensional:

Métodos de BinarySearch

para buscar rápidamente un elemento en un arreglo ordenado.

Métodos IndexOf/LastIndex

para buscar un elemento en arreglos no ordenados.

Métodos Find/FindLast/FindIndex/FindLastIndex/FindAll/Exists/TrueForAll

para buscar elementos en arreglos no ordenados que satisfacen un predicado dado.

Ninguno de los métodos de búsqueda de arreglos lanza una excepción si el valor especificado no se encuentra. En su lugar, si un elemento no se encuentra, los métodos que devuelven un entero retornan -1 (asumiendo un arreglo indexado en cero), y los métodos que devuelven un tipo genérico retornan el valor predeterminado del tipo (por ejemplo, 0 para un entero o null para una cadena).

Los métodos de búsqueda binaria son rápidos, pero solo funcionan en arreglos ordenados y requieren que los elementos sean comparados en términos de orden en lugar de igualdad. Para ello, los métodos de búsqueda binaria pueden aceptar un objeto IComparer o IComparer<T> para arbitrar en decisiones de ordenamiento. Esto debe ser consistente con cualquier comparador utilizado en la ordenación original del arreglo. Si no se proporciona un comparador, se aplicará el algoritmo de ordenamiento predeterminado del tipo en función de su implementación de IComparable / IComparable<T>.

Los métodos IndexOf y LastIndexOf realizan una enumeración simple sobre el arreglo, devolviendo la posición del primer (o último) elemento que coincide con el valor dado.

Los métodos de búsqueda basados en predicados permiten que un delegado de método o una expresión lambda arbitre si un elemento dado es una "coincidencia". Un predicado es simplemente un delegado que acepta un objeto y devuelve true o false:

```
public delegate bool Predicate<T> (T object);
```

En el siguiente ejemplo, buscamos un arreglo de cadenas para encontrar un nombre que contenga la letra "a":

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, ContainsA);
Console.WriteLine (match);     // Jack

ContainsA (string name) { return name.Contains ("a"); }
```

Aquí está el mismo código abreviado con una expresión lambda:

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, n => n.Contains ("a"));     // Jack
```

FindAll devuelve un arreglo con todos los elementos que satisfacen el predicado. De hecho, es equivalente a Enumerable.Where en el espacio de nombres System.Linq, excepto que FindAll devuelve un arreglo de elementos coincidentes en lugar de un IEnumerable<T> del mismo tipo.

Exists devuelve true si algún miembro del arreglo satisface el predicado dado, y es equivalente a Any en System.Linq.Enumerable.

TrueForAll devuelve true si todos los elementos satisfacen el predicado, y es equivalente a All en System.Linq.Enumerable.

Ordenamiento

La clase Array tiene los siguientes métodos de ordenamiento incorporados:

```
// For sorting a single array:

public static void Sort<T> (T[] array);
public static void Sort    (Array array);

// For sorting a pair of arrays:

public static void Sort<TKey,TValue> (TKey[] keys, TValue[] items);
public static void Sort              (Array keys, Array items);
```

Cada uno de estos métodos también está sobrecargado para aceptar lo siguiente adicionalmente:

```
int index                // Starting index at which to begin sorting
int length               // Number of elements to sort
IComparer<T> comparer    // Object making ordering decisions
Comparison<T> comparison // Delegate making ordering decisions
```

El siguiente ejemplo ilustra el uso más sencillo de Sort:

```
int[] numbers = { 3, 2, 1 };
Array.Sort (numbers);                    // Array is now { 1, 2, 3 }
```

Los métodos que aceptan un par de arreglos funcionan reorganizando los elementos de cada arreglo al mismo tiempo, basándose en las decisiones de ordenamiento del primer arreglo. En el siguiente ejemplo, tanto los números como las palabras correspondientes se ordenan en orden numérico:

```
int[] numbers = { 3, 2, 1 };
string[] words = { "three", "two", "one" };
Array.Sort (numbers, words);

// numbers array is now { 1, 2, 3 }
// words   array is now { "one", "two", "three" }
```

Array.Sort requiere que los elementos en el arreglo implementen IComparable (consultar "Comparación de orden" en la página 335). Esto significa que la mayoría de los tipos integrados en C# (como los enteros, como en el ejemplo anterior) pueden ser ordenados. Si los elementos no son intrínsecamente comparables o si deseas anular el ordenamiento predeterminado, debes proporcionar a Sort un proveedor de comparación personalizado que indique la posición relativa de dos elementos. Hay diferentes formas de hacerlo:

- A través de un objeto auxiliar que implemente IComparer/IComparer<T> (consultar "Conexión de igualdad y orden" en la página 388).
- A través de un delegado Comparison:
  ```
  public delegate int Comparison<T> (T x, T y);
  ```

El delegado Comparison sigue la misma semántica que IComparer<T>.CompareTo: si x viene antes que y, se devuelve un entero negativo; si x viene después de y, se devuelve un entero positivo; si x e y tienen la misma posición de ordenamiento, se devuelve 0.

En el siguiente ejemplo, ordenamos un arreglo de enteros de manera que los números impares estén primero:

```
int[] numbers = { 1, 2, 3, 4, 5 };
Array.Sort (numbers, (x, y) => x % 2 == y % 2 ? 0 : x % 2 == 1 ? -1 : 1);

// numbers array is now { 1, 3, 5, 2, 4 }
```



Como alternativa a llamar a Sort, puedes usar los operadores OrderBy y ThenBy de LINQ. A diferencia de Array.Sort, los operadores de LINQ no alteran el arreglo original, sino que emiten el resultado ordenado en una nueva secuencia IEnumerable<T>.

Inversión de elementos

Los siguientes métodos de Array invierten el orden de todos (o una porción) de elementos en un arreglo:

```
public static void Reverse (Array array);
public static void Reverse (Array array, int index, int length);
```

Copia

Array proporciona cuatro métodos para realizar copias superficiales: Clone, CopyTo, Copy y ConstrainedCopy. Los dos primeros son métodos de instancia, mientras que los últimos dos son métodos estáticos.

El método Clone devuelve un nuevo arreglo completamente nuevo (copiado superficialmente). Los métodos CopyTo y Copy copian una porción contigua del arreglo. Copiar un arreglo multidimensional rectangular requiere mapear el índice multidimensional a un índice lineal. Por ejemplo, el cuadrado central (posición [1,1]) en un arreglo de 3x3 se representa con el índice 4, mediante el cálculo: 1 * 3 + 1. Los rangos de origen y destino pueden superponerse sin causar problemas.

ConstrainedCopy realiza una operación atómica: si no se pueden copiar exitosamente todos los elementos solicitados (debido a un error de tipo, por ejemplo), la operación se deshace.

Array también proporciona un método llamado AsReadOnly que devuelve un envoltorio (wrapper) que evita que los elementos sean reasignados.

Conversión y cambio de tamaño

Array.ConvertAll crea y devuelve un nuevo arreglo del tipo TOutput, utilizando el delegado Converter proporcionado para copiar los elementos. El delegado Converter se define de la siguiente manera:

```
public delegate TOutput Converter<TInput,TOutput> (TInput input)
```

El siguiente ejemplo convierte un arreglo de floats en un arreglo de enteros:

```
float[] reals = { 1.3f, 1.5f, 1.8f };
int[] wholes = Array.ConvertAll (reals, r => Convert.ToInt32 (r));

// wholes array is { 1, 2, 2 }
```

El método Resize funciona creando un nuevo arreglo y copiando los elementos, devolviendo el nuevo arreglo a través del parámetro de referencia. Sin embargo, cualquier referencia al arreglo original en otros objetos no se verá afectada.



El espacio de nombres System.Linq ofrece una amplia gama de métodos de extensión adicionales adecuados para la conversión de arreglos. Estos métodos devuelven un IEnumerable<T>, que se puede convertir de nuevo en un arreglo mediante el método ToArray de Enumerable.

Listas, Colas, Pilas y Conjuntos

.NET proporciona un conjunto básico de clases de colección concretas que implementan las interfaces descritas en este capítulo. Esta sección se centra en las colecciones tipo lista (en contraste con las colecciones tipo diccionario, que se tratan en "Diccionarios" en la página 372). Al igual que con las interfaces que discutimos anteriormente, generalmente tienes la opción de versiones genéricas o no genéricas de cada tipo. En términos de flexibilidad y rendimiento, las clases genéricas son superiores, lo que hace que las versiones no genéricas sean redundantes, excepto por compatibilidad con versiones anteriores. Esto difiere de la situación con las interfaces de colección, para las cuales las versiones no genéricas aún son ocasionalmente útiles.

De las clases descritas en esta sección, la clase genérica List es la más comúnmente utilizada.

List<T> y ArrayList

Las clases genéricas List y no genérica ArrayList proporcionan un arreglo de objetos de tamaño dinámico y están entre las más utilizadas de las clases de colección. ArrayList implementa IList, mientras que List<T> implementa tanto IList como IList<T> (y la versión de solo lectura IReadOnlyList<T>). A diferencia de los arreglos, las interfaces de Listas, Colas, Pilas y Conjuntos se implementan públicamente y los métodos como Add y Remove se exponen y funcionan como se esperaría.

Internamente, List<T> y ArrayList funcionan manteniendo un arreglo interno de objetos, el cual es reemplazado por un arreglo más grande cuando se alcanza su capacidad máxima. Agregar elementos al final de la lista es eficiente (ya que generalmente hay un espacio libre al final), pero insertar elementos puede ser lento (ya que todos los elementos después del punto de inserción deben desplazarse para crear un espacio libre), al igual que eliminar elementos (especialmente cerca del inicio). Al igual que con los arreglos, la búsqueda es eficiente si se utiliza el método BinarySearch en una lista que ha sido ordenada, pero de lo contrario es ineficiente ya que cada elemento debe ser verificado individualmente.



List<T> es varias veces más rápido que ArrayList si T es un tipo de valor, debido a que List<T> evita la sobrecarga de encajonamiento y desencajonamiento de elementos.

List<T> y ArrayList proporcionan constructores que aceptan una colección existente de elementos: estos copian cada elemento de la colección existente en la nueva List<T> o ArrayList:

```
public class List<T> : IList<T>, IReadOnlyList<T>
{
  public List ();
  public List (IEnumerable<T> collection);
  public List (int capacity);

  // Add+Insert
  public void Add         (T item);
  public void AddRange    (IEnumerable<T> collection);
  public void Insert      (int index, T item);
  public void InsertRange (int index, IEnumerable<T> collection);

  // Remove
  public bool Remove      (T item);
  public void RemoveAt    (int index);
  public void RemoveRange (int index, int count);
  public int  RemoveAll   (Predicate<T> match);

  // Indexing
  public T this [int index] { get; set; }
  public List<T> GetRange (int index, int count);
  public Enumerator<T> GetEnumerator();

  // Exporting, copying and converting:
  public T[] ToArray();
  public void CopyTo (T[] array);
  public void CopyTo (T[] array, int arrayIndex);
  public void CopyTo (int index, T[] array, int arrayIndex, int count);
  public ReadOnlyCollection<T> AsReadOnly();
  public List<TOutput> ConvertAll<TOutput> (Converter <T,TOutput>
                                            converter);
  // Other:
  public void Reverse();            // Reverses order of elements in list.
  public int Capacity { get;set; }  // Forces expansion of internal array.
```

```
  public void TrimExcess();         // Trims internal array back to size.
  public void Clear();              // Removes all elements, so Count=0.
}

public delegate TOutput Converter <TInput, TOutput> (TInput input);
```

Además de estos miembros, List<T> proporciona versiones de instancia de todos los métodos de búsqueda y ordenación de Array.

El siguiente código demuestra las propiedades y métodos de List (para ejemplos de búsqueda y ordenación, consulta "La clase Array" en la página 355):

```
var words = new List<string>();    // New string-typed list

words.Add ("melon");
words.Add ("avocado");
words.AddRange (new[] { "banana", "plum" } );
words.Insert (0, "lemon");                          // Insert at start
words.InsertRange (0, new[] { "peach", "nashi" });   // Insert at start

words.Remove ("melon");
words.RemoveAt (3);                        // Remove the 4th element
words.RemoveRange (0, 2);                  // Remove first 2 elements

// Remove all strings starting in 'n':
words.RemoveAll (s => s.StartsWith ("n"));

Console.WriteLine (words [0]);                          // first word
Console.WriteLine (words [words.Count - 1]);            // last word
foreach (string s in words) Console.WriteLine (s);      // all words
List<string> subset = words.GetRange (1, 2);            // 2nd->3rd words

string[] wordsArray = words.ToArray();    // Creates a new typed array

// Copy first two elements to the end of an existing array:
string[] existing = new string [1000];
words.CopyTo (0, existing, 998, 2);

List<string> upperCaseWords = words.ConvertAll (s => s.ToUpper());
List<int> lengths = words.ConvertAll (s => s.Length);
```

La clase ArrayList no genérica requiere conversiones incómodas, como se muestra en el siguiente ejemplo:

```
ArrayList al = new ArrayList();
al.Add ("hello");
string first = (string) al [0];
string[] strArr = (string[]) al.ToArray (typeof (string));
```

Estas conversiones no se pueden verificar por el compilador. El siguiente código se compila correctamente pero falla en tiempo de ejecución:

```
int first = (int) al [0];    // Runtime exception
```



Un ArrayList es funcionalmente similar a List<object>. Ambos son útiles cuando necesitas una lista de elementos de tipos mixtos que no comparten ningún tipo base común, aparte de object. Una posible ventaja de elegir un ArrayList en este caso sería si necesitas manipular la lista utilizando reflexión (Capítulo 19). La reflexión es más fácil con un ArrayList no genérico que con un List<object>.

Si importas el espacio de nombres System.Linq, puedes convertir un ArrayList en un List genérico llamando a los métodos Cast y luego ToList:

```
ArrayList al = new ArrayList();
al.AddRange (new[] { 1, 5, 9 } );
List<int> list = al.Cast<int>().ToList();
```

Cast y ToList son métodos de extensión en la clase System.Linq.Enumerable.

LinkedList<T>

LinkedList<T> es una lista enlazada doblemente enlazada (ver Figura 7-4). Una lista enlazada doblemente enlazada es una cadena de nodos en la que cada nodo hace referencia al nodo anterior, al nodo siguiente y al elemento actual. Su principal ventaja es que un elemento siempre se puede insertar de manera eficiente en cualquier lugar de la lista, ya que solo implica crear un nuevo nodo y actualizar algunas referencias. Sin embargo, encontrar dónde insertar el nodo en primer lugar puede ser lento, ya que no hay un mecanismo intrínseco para indexar directamente en una lista enlazada; cada nodo debe ser recorrido y no es posible realizar búsquedas binarias.

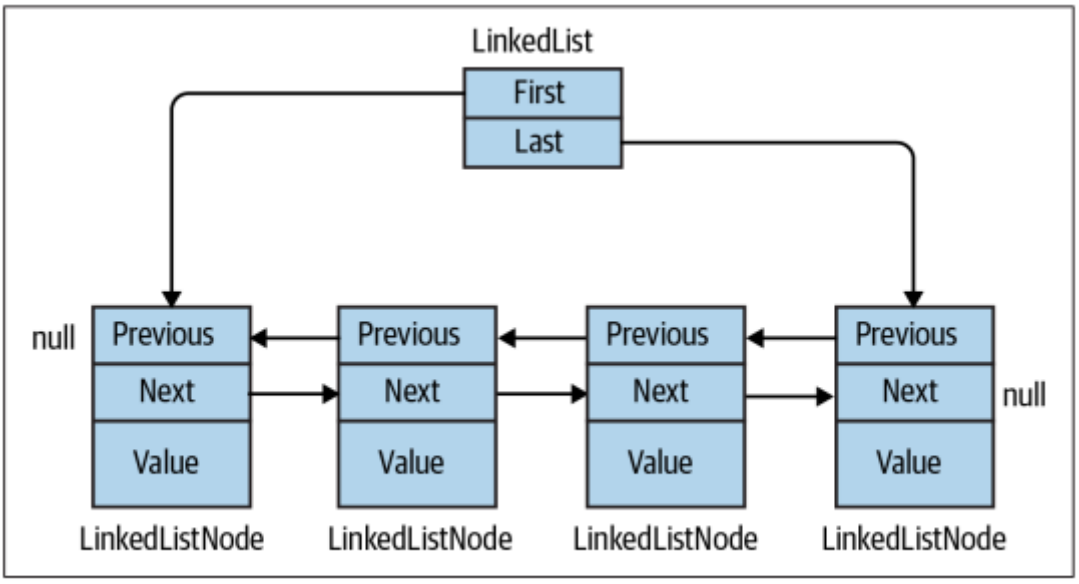


Figure 7-4. LinkedList<T>

LinkedList<T> implementa IEnumerable<T> e ICollection<T> (y sus versiones no genéricas), pero no implementa IList<T> porque no se admite el acceso mediante índices. Los nodos de la lista se implementan a través de la siguiente clase:

```
public sealed class LinkedListNode<T>
{
  public LinkedList<T> List { get; }
  public LinkedListNode<T> Next { get; }
  public LinkedListNode<T> Previous { get; }
  public T Value { get; set; }
}
```

Cuando se agrega un nodo, se puede especificar su posición relativa a otro nodo o al inicio/final de la lista. LinkedList<T> proporciona los siguientes métodos para esto:

```
public void AddFirst(LinkedListNode<T> node);
public LinkedListNode<T> AddFirst (T value);

public void AddLast (LinkedListNode<T> node);
public LinkedListNode<T> AddLast (T value);

public void AddAfter (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddAfter (LinkedListNode<T> node, T value);

public void AddBefore (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddBefore (LinkedListNode<T> node, T value);
```

Se proporcionan métodos similares para eliminar elementos:

```
public void Clear();

public void RemoveFirst();
public void RemoveLast();

public bool Remove (T value);
public void Remove (LinkedListNode<T> node);
```

LinkedList<T> tiene campos internos para realizar un seguimiento del número de elementos en la lista, así como de la cabeza y la cola de la lista. Estos se exponen en las siguientes propiedades públicas:

```
public int Count { get; }                    // Fast
public LinkedListNode<T> First { get; }      // Fast
public LinkedListNode<T> Last { get; }       // Fast
```

LinkedList<T> también admite los siguientes métodos de búsqueda (cada uno requiere que la lista sea enumerada internamente):

```
public bool Contains (T value);
public LinkedListNode<T> Find (T value);
public LinkedListNode<T> FindLast (T value);
```

Finalmente, LinkedList<T> admite la copia a un arreglo para procesamiento indexado y obtener un enumerador para admitir la instrucción foreach:

```
public void CopyTo (T[] array, int index);
public Enumerator<T> GetEnumerator();
```

Aquí tienes una demostración del uso de LinkedList<string>:

```
var tune = new LinkedList<string>();
tune.AddFirst ("do");                           // do
tune.AddLast ("so");                            // do - so

tune.AddAfter (tune.First, "re");               // do - re- so
tune.AddAfter (tune.First.Next, "mi");          // do - re - mi- so
tune.AddBefore (tune.Last, "fa");               // do - re - mi - fa- so

tune.RemoveFirst();                             // re - mi - fa - so
tune.RemoveLast();                              // re - mi - fa

LinkedListNode<string> miNode = tune.Find ("mi");
tune.Remove (miNode);                           // re - fa
tune.AddFirst (miNode);                         // mi- re - fa

foreach (string s in tune) Console.WriteLine (s);
```

Queue<T> y Queue

Queue<T> y Queue son estructuras de datos de tipo first-in, first-out (FIFO), que proporcionan métodos para Enqueue (agregar un elemento al final de la cola) y Dequeue (recuperar y eliminar el elemento en la cabeza de la cola). También se proporciona un método Peek para devolver el elemento en la cabeza de la cola sin eliminarlo, así como una propiedad Count (útil para verificar si hay elementos antes de hacer dequeue).

Aunque las colas son enumerables, no implementan IList<T>/IList porque los elementos no se pueden acceder directamente por índice. Sin embargo, se proporciona un método ToArray para copiar los elementos a un arreglo desde el cual se pueden acceder aleatoriamente:

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
  public Queue();
  public Queue (IEnumerable<T> collection);   // Copies existing elements
  public Queue (int capacity);                // To lessen auto-resizing
  public void Clear();
  public bool Contains (T item);
  public void CopyTo (T[] array, int arrayIndex);
  public int Count { get; }
  public T Dequeue();
  public void Enqueue (T item);
  public Enumerator<T> GetEnumerator();       // To support foreach
  public T Peek();
  public T[] ToArray();
  public void TrimExcess();
}
```

A continuación, se muestra un ejemplo de uso de Queue<int>:

```
var q = new Queue<int>();
q.Enqueue (10);
q.Enqueue (20);
```

```
int[] data = q.ToArray();         // Exports to an array
Console.WriteLine (q.Count);      // "2"
Console.WriteLine (q.Peek());     // "10"
Console.WriteLine (q.Dequeue());  // "10"
Console.WriteLine (q.Dequeue());  // "20"
Console.WriteLine (q.Dequeue());  // throws an exception (queue empty)
```

Las colas (Queues) se implementan internamente utilizando un array que se redimensiona según sea necesario, de manera similar a la clase genérica List. La cola mantiene índices que apuntan directamente a los elementos de la cabeza y la cola; por lo tanto, las operaciones de encolar (Enqueue) y desencolar (Dequeue) son extremadamente rápidas (excepto cuando se requiere una redimensión interna).

Stack<T> y Stack

Las pilas (Stack<T>) son estructuras de datos de último en entrar, primero en salir (LIFO), que proporcionan métodos para empujar (Push) un elemento en la parte superior de la pila y sacar (Pop) un elemento de la parte superior de la pila. También se proporciona un método Peek no destructivo, así como una propiedad Count y un método ToArray para exportar los datos para acceso aleatorio:

```
public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
  public Stack();
  public Stack (IEnumerable<T> collection);   // Copies existing elements
  public Stack (int capacity);                // Lessens auto-resizing
  public void Clear();
  public bool Contains (T item);
  public void CopyTo (T[] array, int arrayIndex);
  public int Count { get; }
  public Enumerator<T> GetEnumerator();       // To support foreach
  public T Peek();
  public T Pop();
  public void Push (T item);
  public T[] ToArray();
  public void TrimExcess();
}
```

Aquí tienes un ejemplo que demuestra el uso de Stack<int>:

```
var s = new Stack<int>();
s.Push (1);                     //           Stack = 1
s.Push (2);                     //           Stack = 1,2
s.Push (3);                     //           Stack = 1,2,3
Console.WriteLine (s.Count);    // Prints 3
Console.WriteLine (s.Peek());   // Prints 3,  Stack = 1,2,3
Console.WriteLine (s.Pop());    // Prints 3,  Stack = 1,2
Console.WriteLine (s.Pop());    // Prints 2,  Stack = 1
Console.WriteLine (s.Pop());    // Prints 1,  Stack = <empty>
Console.WriteLine (s.Pop());    // throws exception
```

Las pilas se implementan internamente con un array que se redimensiona según sea necesario, al igual que Queue<T> y List<T>.

BitArray

Un BitArray es una colección de tamaño dinámico de valores booleanos compactados. Es más eficiente en términos de memoria que un simple array de bool o una lista genérica de bool, ya que utiliza solo un bit para cada valor, mientras que el tipo bool normalmente ocupa un byte para cada valor.

El indexador de BitArray lee y escribe bits individuales:

```
var bits = new BitArray(2);
bits[1] = true;
```

Existen cuatro métodos de operadores bit a bit (And, Or, Xor y Not). Todos excepto el último aceptan otro BitArray:

```
bits.Xor (bits);               // Bitwise exclusive-OR bits with itself
Console.WriteLine (bits[1]);   // False
```

HashSet<T> y SortedSet<T>

HashSet<T> y SortedSet<T> tienen las siguientes características distintivas:

- Sus métodos Contains se ejecutan rápidamente utilizando una búsqueda basada en hash.
- No almacenan elementos duplicados y ignoran silenciosamente las solicitudes para agregar duplicados.
- No se puede acceder a un elemento por posición.

SortedSet<T> mantiene los elementos en orden, mientras que HashSet<T> no lo hace.

La similitud entre HashSet<T> y SortedSet<T> se captura mediante la interfaz ISet<T>. A partir de .NET 5, estas clases también implementan una interfaz llamada IReadOnlySet<T>, que también es implementada por los tipos de conjuntos inmutables (consultar "Colecciones inmutables" en la página 384).

HashSet<T> se implementa con una tabla hash que almacena solo claves; SortedSet<T> se implementa con un árbol rojo/negro.

Ambas colecciones implementan ICollection<T> y ofrecen métodos como Contains, Add y Remove. Además, existe un método de eliminación basado en predicado llamado RemoveWhere.

El siguiente ejemplo crea un HashSet<char> a partir de una colección existente, verifica la pertenencia de elementos y luego enumera la colección (observa la ausencia de duplicados):

```
var letters = new HashSet<char> ("the quick brown fox");

Console.WriteLine (letters.Contains ('t'));      // true
Console.WriteLine (letters.Contains ('j'));      // false

foreach (char c in letters) Console.Write (c);   // the quickbrownfx
```

(La razón por la que podemos pasar un string al constructor de HashSet<char> es porque string implementa IEnumerable<char>.)

Los métodos realmente interesantes son las operaciones de conjunto. Las siguientes operaciones de conjunto son destructivas en el sentido de que modifican el conjunto:

```
public void UnionWith           (IEnumerable<T> other);   // Adds
public void IntersectWith       (IEnumerable<T> other);   // Removes
public void ExceptWith          (IEnumerable<T> other);   // Removes
public void SymmetricExceptWith (IEnumerable<T> other);   // Removes
```

Mientras que los siguientes métodos simplemente consultan el conjunto y son no destructivos:

```
public bool IsSubsetOf         (IEnumerable<T> other);
public bool IsProperSubsetOf   (IEnumerable<T> other);
public bool IsSupersetOf       (IEnumerable<T> other);
public bool IsProperSupersetOf (IEnumerable<T> other);
public bool Overlaps           (IEnumerable<T> other);
public bool SetEquals          (IEnumerable<T> other);
```

UnionWith agrega todos los elementos del segundo conjunto al conjunto original (excluyendo duplicados). IntersectWith elimina los elementos que no están en ambos conjuntos. Podemos extraer todas las vocales de nuestro conjunto de caracteres de la siguiente manera:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.IntersectWith ("aeiou");
foreach (char c in letters) Console.Write (c);     // euio
```

ExceptWith elimina los elementos especificados del conjunto fuente. Aquí eliminamos todas las vocales del conjunto:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.ExceptWith ("aeiou");
foreach (char c in letters) Console.Write (c);     // th qckbrwnfx
```

SymmetricExceptWith elimina todos los elementos excepto aquellos que son únicos en uno u otro conjunto:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.SymmetricExceptWith ("the lazy brown fox");
foreach (char c in letters) Console.Write (c);     // quicklazy
```

Ten en cuenta que debido a que HashSet<T> y SortedSet<T> implementan IEnumerable<T>, puedes usar otro tipo de conjunto (o colección) como argumento para cualquiera de los métodos de operaciones de conjunto.

SortedSet<T> ofrece todos los miembros de HashSet<T>, además de lo siguiente:

```
public virtual SortedSet<T> GetViewBetween (T lowerValue, T upperValue)
public IEnumerable<T> Reverse()
public T Min { get; }
public T Max { get; }
```

SortedSet<T> también acepta un IComparer<T> opcional en su constructor (en lugar de un comparador de igualdad).

Aquí tienes un ejemplo de carga de las mismas letras en un SortedSet<char>:

```
var letters = new SortedSet<char> ("the quick brown fox");
foreach (char c in letters) Console.Write (c);   //  bcefhiknoqrtuwx
```

Siguiendo con esto, podemos obtener las letras en el conjunto entre 'f' e 'i' de la siguiente manera:

```
foreach (char c in letters.GetViewBetween ('f', 'i'))
  Console.Write (c);                                    //  fhi
```

Diccionarios

Un diccionario es una colección en la que cada elemento es un par clave/valor. Los diccionarios se utilizan principalmente para búsquedas y listas ordenadas.

.NET define un protocolo estándar para diccionarios a través de las interfaces IDictionary e IDictionary<TKey, TValue>, así como un conjunto de clases de diccionario de propósito general. Las clases difieren en los siguientes aspectos:

- Si los elementos se almacenan en secuencia ordenada.
- Si los elementos se pueden acceder por posición (índice) además de por clave.
- Si son genéricos o no genéricos.
- Si es rápido o lento recuperar elementos por clave en un diccionario grande.

La Tabla 7-1 resume cada una de las clases de diccionario y cómo difieren en estos aspectos.

Los tiempos de rendimiento están en milisegundos y se basan en la realización de 50,000 operaciones en un diccionario con claves y valores enteros en un PC de 1.5 GHz. (Las diferencias de rendimiento entre las contrapartes genéricas y no genéricas que utilizan la misma estructura de colección subyacente se deben al boxing y solo se muestran con elementos de tipo de valor.)

Table 7-1. Dictionary classes

Type	Internal structure	Retrieve by index?	Memory overhead (avg. bytes per item)	Speed: random insertion	Speed: sequential insertion	Speed: retrieval by key
Unsorted						
`Dictionary <K,V>`	Hashtable	No	22	30	30	20
`Hashtable`	Hashtable	No	38	50	50	30
`ListDictionary`	Linked list	No	36	50,000	50,000	50,000
`OrderedDictionary`	Hashtable + array	Yes	59	70	70	40

Type	Internal structure	Retrieve by index?	Memory overhead (avg. bytes per item)	Speed: random insertion	Speed: sequential insertion	Speed: retrieval by key
Sorted						
`SortedDictionary <K,V>`	Red/black tree	No	20	130	100	120
`SortedList <K,V>`	2xArray	Yes	2	3,300	30	40
`SortedList`	2xArray	Yes	27	4,500	100	180

En la notación Big-O, el tiempo de recuperación por clave es el siguiente:

- O(1) para Hashtable, Dictionary y OrderedDictionary.
- O(log n) para SortedDictionary y SortedList.
- O(n) para ListDictionary (y tipos no diccionario como List<T>).

Donde "n" es el número de elementos en la colección.

La interfaz IDictionary<TKey, TValue>

La interfaz IDictionary<TKey, TValue> define el protocolo estándar para todas las colecciones basadas en clave/valor. Extiende ICollection<T> al agregar métodos y propiedades para acceder a elementos basados en una clave de tipo arbitrario:

```
public interface IDictionary <TKey, TValue> :
  ICollection <KeyValuePair <TKey, TValue>>, IEnumerable
{
   bool ContainsKey (TKey key);
   bool TryGetValue (TKey key, out TValue value);
   void Add         (TKey key, TValue value);
   bool Remove      (TKey key);

   TValue this [TKey key]      { get; set; }  // Main indexer - by key
   ICollection <TKey> Keys     { get; }       // Returns just keys
   ICollection <TValue> Values { get; }       // Returns just values
}
```



También existe una interfaz llamada IReadOnlyDictionary<TKey, TValue>, que define el subconjunto de solo lectura de los miembros del diccionario.

Para agregar un elemento a un diccionario, puedes llamar a Add o usar el accesor de índice (set) del diccionario; este último agrega un elemento al diccionario si la clave aún no está presente (o actualiza el elemento si está presente). Las implementaciones de diccionario prohíben las claves duplicadas, por lo que llamar a Add dos veces con la misma clave lanzará una excepción.

Para recuperar un elemento de un diccionario, puedes utilizar tanto el indexador como el método TryGetValue. Si la clave no existe, el indexador lanzará una excepción, mientras que TryGetValue devolverá false. Puedes comprobar la pertenencia a través del método ContainsKey; sin embargo, esto conlleva el costo de dos búsquedas si posteriormente recuperas el elemento.

Al enumerar directamente sobre un IDictionary<TKey, TValue>, obtendrás una secuencia de estructuras KeyValuePair:

```
public struct KeyValuePair <TKey, TValue>
{
  public TKey Key     { get; }
  public TValue Value { get; }
}
```

Puedes enumerar solo las claves o los valores a través de las propiedades Keys/Values del diccionario.

Demostraremos el uso de esta interfaz con la clase genérica Dictionary en la siguiente sección.

IDictionary

La interfaz no genérica IDictionary es, en principio, igual a IDictionary<TKey, TValue>, excepto por dos diferencias funcionales importantes. Es importante tener en cuenta estas diferencias, ya que IDictionary se encuentra en código heredado (incluido el BCL de .NET en algunos lugares):

- Recuperar una clave inexistente a través del indexador devuelve null en lugar de lanzar una excepción.
- Contains verifica la pertenencia en lugar de ContainsKey.

Al enumerar sobre un IDictionary no genérico, obtendrás una secuencia de estructuras DictionaryEntry:

```
public struct DictionaryEntry
{
  public object Key   { get; set; }
  public object Value { get; set; }
}
```

Dictionary<TKey, TValue> y Hashtable

La clase genérica Dictionary es una de las colecciones más utilizadas (junto con la colección List<T>). Utiliza una estructura de datos de tabla hash para almacenar claves y valores, y es rápida y eficiente.



La versión no genérica de Dictionary<TKey, TValue> se llama Hashtable; no existe una clase no genérica llamada Dictionary. Cuando nos referimos simplemente a Dictionary, nos referimos a la clase genérica Dictionary<TKey, TValue>.

El diccionario implementa tanto las interfaces genéricas como las no genéricas de IDictionary, siendo la interfaz genérica IDictionary la que se expone públicamente. El diccionario es, de hecho, una implementación "clásica" de IDictionary.

Aquí tienes un ejemplo de cómo usarlo:

```
var d = new Dictionary<string, int>();

d.Add("One", 1);
d["Two"] = 2;      // adds to dictionary because "two" not already present
d["Two"] = 22;     // updates dictionary because "two" is now present
d["Three"] = 3;

Console.WriteLine (d["Two"]);                // Prints "22"
Console.WriteLine (d.ContainsKey ("One"));   // true (fast operation)
Console.WriteLine (d.ContainsValue (3));     // true (slow operation)
int val = 0;
if (!d.TryGetValue ("onE", out val))
  Console.WriteLine ("No val");              // "No val" (case sensitive)

// Three different ways to enumerate the dictionary:

foreach (KeyValuePair<string, int> kv in d)          //  One; 1
  Console.WriteLine (kv.Key + "; " + kv.Value);      //  Two; 22
                                                     //  Three; 3

foreach (string s in d.Keys) Console.Write (s);      // OneTwoThree
Console.WriteLine();
foreach (int i in d.Values) Console.Write (i);       // 1223
```

La estructura subyacente del diccionario utiliza una tabla hash convirtiendo la clave de cada elemento en un valor hash de tipo entero (pseudo-único) y luego aplica un algoritmo para convertir ese valor hash en una clave hash. Esta clave hash se utiliza internamente para determinar en qué "cubo" pertenece una entrada. Si un cubo contiene más de un valor, se realiza una búsqueda lineal en el cubo. Una buena función hash no se esfuerza por devolver valores hash estrictamente únicos (lo cual generalmente sería imposible), sino que busca devolver valores hash distribuidos de manera uniforme en el espacio de enteros de 32 bits. Esto evita el escenario de terminar con unos pocos cubos muy grandes e ineficientes.

Un diccionario puede trabajar con claves de cualquier tipo, siempre y cuando pueda determinar la igualdad entre las claves y obtener sus valores hash. De forma predeterminada, la igualdad se determina mediante el método object.Equals de la clave, y el valor hash pseudo-único se obtiene mediante el método GetHashCode de la clave. Puedes cambiar este comportamiento ya sea sobrescribiendo estos métodos o proporcionando un objeto IEqualityComparer al construir el diccionario. Una aplicación común de esto es especificar un comparador de igualdad que ignore las diferencias de mayúsculas y minúsculas al usar claves de tipo string:

```
var d = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

Se discute esto con más detalle en la sección "Plugging in Equality and Order" en la página 388.

Al igual que con muchos otros tipos de colecciones, puedes mejorar ligeramente el rendimiento de un diccionario especificando el tamaño esperado de la colección en el constructor, evitando o reduciendo la necesidad de operaciones internas de cambio de tamaño.

La versión no genérica se llama Hashtable y es funcionalmente similar, excepto por las diferencias derivadas de exponer la interfaz no genérica IDictionary discutida anteriormente.

La desventaja de Dictionary y Hashtable es que los elementos no están ordenados. Además, no se conserva el orden original en el que se agregaron los elementos. Como ocurre con todos los diccionarios, no se permiten claves duplicadas.


Cuando se introdujeron las colecciones genéricas en 2005, el equipo de CLR decidió nombrarlas según lo que representan (Dictionary, List) en lugar de cómo se implementan internamente (Hashtable, ArrayList). Aunque esto es bueno porque les da la libertad de cambiar la implementación más adelante, también significa que el contrato de rendimiento (a menudo el criterio más importante al elegir un tipo de colección sobre otro) ya no se refleja en el nombre.

OrderedDictionary

OrderedDictionary es un diccionario no genérico que mantiene los elementos en el mismo orden en el que se agregaron. Con un OrderedDictionary, puedes acceder a los elementos tanto por índice como por clave.


Un OrderedDictionary no es un diccionario ordenado.

Un OrderedDictionary es una combinación de un Hashtable y un ArrayList. Esto significa que tiene toda la funcionalidad de un Hashtable, además de funciones como RemoveAt y un indexador entero. También expone las propiedades Keys y Values que devuelven los elementos en su orden original.

Esta clase se introdujo en .NET 2.0, sin embargo, curiosamente, no hay una versión genérica.

ListDictionary y HybridDictionary

ListDictionary utiliza una lista enlazada simple para almacenar los datos subyacentes. No proporciona ordenamiento, aunque conserva el orden de entrada original de los elementos. ListDictionary es extremadamente lento con listas grandes. Su única "reclamación a la fama" real es su eficiencia con listas muy pequeñas (menos de 10 elementos).

HybridDictionary es un ListDictionary que se convierte automáticamente en un Hashtable al alcanzar un cierto tamaño, para abordar los problemas de rendimiento de ListDictionary. La idea es tener una huella de memoria baja cuando el diccionario es pequeño y un buen rendimiento cuando el diccionario es grande. Sin embargo, considerando el costo de convertir de una estructura a otra, y el hecho de que un Dictionary no es excesivamente pesado o lento en ninguno de los escenarios, no sufrirías de forma irrazonable al utilizar un Dictionary desde el principio.

Ambas clases solo vienen en forma no genérica.

Diccionarios ordenados

La Biblioteca de Clases Base de .NET proporciona dos clases de diccionarios estructuradas internamente de manera que su contenido siempre esté ordenado por clave:

- SortedDictionary<TKey, TValue>
- SortedList<TKey, TValue>1

En esta sección, abreviamos <TKey, TValue> a <,>.

SortedDictionary<,> utiliza un árbol rojo/negro: una estructura de datos diseñada para funcionar de manera consistente en cualquier escenario de inserción o recuperación.

SortedList<,> se implementa internamente con un par de matrices ordenadas, lo que proporciona una recuperación rápida (a través de una búsqueda binaria) pero un rendimiento de inserción deficiente (debido a que los valores existentes deben desplazarse para hacer espacio para una nueva entrada).

SortedDictionary<,> es mucho más rápido que SortedList<,> al insertar elementos en una secuencia aleatoria (especialmente con listas grandes). Sin embargo, SortedList<,> tiene una capacidad adicional: acceder a los elementos tanto por índice como por clave. Con una lista ordenada, puedes ir directamente al enésimo elemento en la secuencia de ordenación (a través del indexador en las propiedades Keys/Values). Para hacer lo mismo con un SortedDictionary<,>, debes enumerar manualmente los n elementos. (Alternativamente, podrías escribir una clase que combine un diccionario ordenado con una lista).

Ninguna de las tres colecciones permite claves duplicadas (como es el caso con todos los diccionarios).

El siguiente ejemplo utiliza reflexión para cargar todos los métodos definidos en System.Object en un diccionario ordenado por nombre y luego enumera sus claves y valores:

```
// MethodInfo is in the System.Reflection namespace

var sorted = new SortedList <string, MethodInfo>();

foreach (MethodInfo m in typeof (object).GetMethods())
  sorted [m.Name] = m;

foreach (string name in sorted.Keys)
  Console.WriteLine (name);

foreach (MethodInfo m in sorted.Values)
  Console.WriteLine (m.Name + " returns a " + m.ReturnType);
```

Aquí está el resultado de la primera enumeración:

```
Equals
GetHashCode
GetType
ReferenceEquals
ToString
```

Aquí está el resultado de la segunda enumeración:

```
Equals returns a System.Boolean
GetHashCode returns a System.Int32
GetType returns a System.Type
ReferenceEquals returns a System.Boolean
ToString returns a System.String
```

Observa que hemos poblado el diccionario a través de su indexador. Si en su lugar hubiéramos utilizado el método Add, lanzaría una excepción porque la clase objeto sobre la que estamos reflexionando sobrecarga el método Equals y no se pueden agregar claves duplicadas a un diccionario. Al utilizar el indexador, la entrada posterior sobrescribe la entrada anterior, evitando este error.



Puedes almacenar múltiples elementos con la misma clave haciendo que cada elemento de valor sea una lista:

```
SortedList <string, List<MethodInfo>>
```

Ampliando nuestro ejemplo, lo siguiente recupera el MethodInfo cuya clave es "GetHashCode", al igual que con un diccionario ordinario:

```
Console.WriteLine (sorted ["GetHashCode"]);      // Int32 GetHashCode()
```

Hasta ahora, todo lo que hemos hecho también funcionaría con un SortedDictionary<,>. Sin embargo, las dos siguientes líneas, que recuperan la última clave y valor, solo funcionan con una lista ordenada:

```
Console.WriteLine (sorted.Keys  [sorted.Count - 1]);           // ToString
Console.WriteLine (sorted.Values[sorted.Count - 1].IsVirtual); // True
```

Colecciones personalizables y proxies

Las clases de colección discutidas en secciones anteriores son convenientes porque puedes instanciarlas directamente, pero no te permiten controlar lo que sucede cuando se agrega o se elimina un elemento de la colección. Con colecciones fuertemente tipadas en una aplicación, a veces necesitas este control. Por ejemplo:

- Para lanzar un evento cuando se agrega o se elimina un elemento.
- Para actualizar propiedades debido al elemento agregado o eliminado.
- Para detectar una operación de agregado o eliminación "no permitida" y lanzar una excepción (por ejemplo, si la operación viola una regla de negocio)

La BCL de .NET proporciona clases de colección con este propósito exacto en el espacio de nombres System.Collections.ObjectModel. Estas son esencialmente proxies o envoltorios que implementan IList<T> o IDictionary<,> al reenviar los métodos a través de una colección subyacente. Cada operación de agregar, eliminar o limpiar se enruta a través de un método virtual que actúa como una "puerta de entrada" cuando se anula.

Las clases de colección personalizables se utilizan comúnmente para colecciones expuestas públicamente, por ejemplo, una colección de controles expuestos públicamente en una clase System.Windows.Form.

Collection<T> y CollectionBase

La clase Collection<T> es un envoltorio personalizable para List<T>.

Además de implementar IList<T> e IList, define cuatro métodos virtuales adicionales y una propiedad protegida de la siguiente manera:

```
public class Collection<T> :
  IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
{
   // ...

   protected virtual void ClearItems();
   protected virtual void InsertItem (int index, T item);
   protected virtual void RemoveItem (int index);
   protected virtual void SetItem (int index, T item);

   protected IList<T> Items { get; }
}
```

Los métodos virtuales proporcionan la puerta de entrada mediante la cual puedes "conectarte" para cambiar o mejorar el comportamiento normal de la lista. La propiedad protegida Items permite al implementador acceder directamente a la "lista interna"; esto se utiliza para realizar cambios internos sin que los métodos virtuales se activen.

Los métodos virtuales no necesitan ser anulados; se pueden dejar sin cambios hasta que haya un requisito para alterar el comportamiento predeterminado de la lista. El siguiente ejemplo muestra el uso típico de "esqueleto" de Collection<T>:

```
Zoo zoo = new Zoo();
zoo.Animals.Add (new Animal ("Kangaroo", 10));
zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
foreach (Animal a in zoo.Animals) Console.WriteLine (a.Name);

public class Animal
{
  public string Name;
  public int Popularity;

  public Animal (string name, int popularity)
```

```
  {
    Name = name; Popularity = popularity;
  }
}

public class AnimalCollection : Collection <Animal>
{
  // AnimalCollection is already a fully functioning list of animals.
  // No extra code is required.
}

public class Zoo   // The class that will expose AnimalCollection.
{                  // This would typically have additional members.

  public readonly AnimalCollection Animals = new AnimalCollection();
}
```

Como está, AnimalCollection no es más funcional que una simple List<Animal>; su función es proporcionar una base para futuras extensiones. Para ilustrar esto, agreguemos ahora una propiedad Zoo a Animal para que pueda hacer referencia al zoológico en el que vive y anule cada uno de los métodos virtuales en Collection<Animal> para mantener automáticamente esa propiedad:

```
public class Animal
{
  public string Name;
  public int Popularity;
  public Zoo Zoo { get; internal set; }
  public Animal(string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}

public class AnimalCollection : Collection <Animal>
{
  Zoo zoo;
  public AnimalCollection (Zoo zoo) { this.zoo = zoo; }

  protected override void InsertItem (int index, Animal item)
  {
    base.InsertItem (index, item);
    item.Zoo = zoo;
  }
  protected override void SetItem (int index, Animal item)
  {
    base.SetItem (index, item);
    item.Zoo = zoo;
  }
  protected override void RemoveItem (int index)
  {
    this [index].Zoo = null;
    base.RemoveItem (index);
  }
  protected override void ClearItems()
```

```
  {
    foreach (Animal a in this) a.Zoo = null;
    base.ClearItems();
  }
}

public class Zoo
{
  public readonly AnimalCollection Animals;
  public Zoo() { Animals = new AnimalCollection (this); }
}
```

Collection<T> también tiene un constructor que acepta una IList<T> existente. A diferencia de otras clases de colección, la lista suministrada se proxifica en lugar de copiarse, lo que significa que los cambios posteriores se reflejarán en la colección Collection<T> envolvente (aunque sin que se activen los métodos virtuales de Collection<T>). Por otro lado, los cambios realizados a través de Collection<T> modificarán la lista subyacente.

CollectionBase

CollectionBase es la versión no genérica de Collection<T>. Proporciona la mayoría de las mismas características que Collection<T>, pero es más incómodo de usar. En lugar de los métodos de plantilla InsertItem, RemoveItem, SetItem y ClearItem, CollectionBase tiene métodos de "gancho" que duplican la cantidad de métodos requeridos: OnInsert, OnInsertComplete, OnSet, OnSetComplete, OnRemove, OnRemoveComplete, OnClear y OnClearComplete. Debido a que CollectionBase no es genérico, también debes implementar métodos tipados al heredar de él, al menos un indexador tipado y un método Add.

KeyedCollection<TKey,TItem> y DictionaryBase

KeyedCollection<TKey,TItem> hereda de Collection<TItem>. Añade y sustrae funcionalidad. Lo que agrega es la capacidad de acceder a los elementos por clave, al igual que en un diccionario. Lo que resta es la capacidad de proxificar tu propia lista interna.

Una colección indexada se asemeja a un OrderedDictionary en el sentido de que combina una lista lineal con una tabla hash. Sin embargo, a diferencia de OrderedDictionary, no implementa IDictionary y no admite el concepto de par clave/valor. En su lugar, las claves se obtienen de los propios elementos a través del método abstracto GetKeyForItem. Esto significa que enumerar una colección indexada es como enumerar una lista ordinaria.

La mejor forma de pensar en KeyedCollection<TKey,TItem> es como una Collection<TItem> con búsqueda rápida por clave.

Debido a que hereda de Collection<>, una colección indexada hereda todas las funcionalidades de Collection<>, excepto la capacidad de especificar una lista existente en la construcción. Los miembros adicionales que define son los siguientes:

```
public abstract class KeyedCollection <TKey, TItem> : Collection <TItem>
```

```
  // ...

  protected abstract TKey GetKeyForItem(TItem item);
  protected void ChangeItemKey(TItem item, TKey newKey);

  // Fast lookup by key - this is in addition to lookup by index.
  public TItem this[TKey key] { get; }

  protected IDictionary<TKey, TItem> Dictionary { get; }
}
```

GetKeyForItem es el método que el implementador sobrescribe para obtener la clave de un elemento a partir del objeto subyacente. El método ChangeItemKey debe ser llamado si la propiedad de clave del elemento cambia, para actualizar el diccionario interno. La propiedad Dictionary devuelve el diccionario interno utilizado para implementar la búsqueda, el cual se crea cuando se agrega el primer elemento. Este comportamiento puede cambiarse especificando un umbral de creación en el constructor, retrasando la creación del diccionario interno hasta que se alcance el umbral (mientras tanto, se realiza una búsqueda lineal si se solicita un elemento por clave). Una buena razón para no especificar un umbral de creación es que tener un diccionario válido puede ser útil para obtener una ICollection<> de claves a través de la propiedad Keys del diccionario. Esta colección luego se puede pasar a una propiedad pública.

El uso más común de KeyedCollection<,> es proporcionar una colección de elementos accesibles tanto por índice como por nombre. Para demostrar esto, revisemos el ejemplo del zoológico, esta vez implementando AnimalCollection como KeyedCollection<string, Animal>:

```
public class Animal
{
  string name;
  public string Name
  {
    get { return name; }
    set {
      if (Zoo != null) Zoo.Animals.NotifyNameChange (this, value);
      name = value;
    }
  }
  public int Popularity;
  public Zoo Zoo { get; internal set; }

  public Animal (string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}

public class AnimalCollection : KeyedCollection <string, Animal>
{
  Zoo zoo;
  public AnimalCollection (Zoo zoo) { this.zoo = zoo; }
```

```
  internal void NotifyNameChange (Animal a, string newName) =>
    this.ChangeItemKey (a, newName);

  protected override string GetKeyForItem (Animal item) => item.Name;

  // The following methods would be implemented as in the previous example
  protected override void InsertItem (int index, Animal item)...
  protected override void SetItem (int index, Animal item)...
  protected override void RemoveItem (int index)...
  protected override void ClearItems()...
}

public class Zoo
{
  public readonly AnimalCollection Animals;
  public Zoo() { Animals = new AnimalCollection (this); }
}
```

El siguiente código demuestra su uso:

```
Zoo zoo = new Zoo();
zoo.Animals.Add (new Animal ("Kangaroo", 10));
zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
Console.WriteLine (zoo.Animals [0].Popularity);               // 10
Console.WriteLine (zoo.Animals ["Mr Sea Lion"].Popularity);   // 20
zoo.Animals ["Kangaroo"].Name = "Mr Roo";
Console.WriteLine (zoo.Animals ["Mr Roo"].Popularity);        // 10
```

DictionaryBase

La versión no genérica de KeyedCollection se llama DictionaryBase. Esta clase heredada toma un enfoque muy diferente al implementar IDictionary y utiliza métodos de "hook" similares a CollectionBase: OnInsert, OnInsertComplete, OnSet, OnSetComplete, OnRemove, OnRemoveComplete, OnClear y OnClearComplete (y además, OnGet). La principal ventaja de implementar IDictionary en lugar de utilizar el enfoque de KeyedCollection es que no es necesario heredar de ella para obtener claves. Sin embargo, dado que el propósito mismo de DictionaryBase es ser subclasificado, esto no supone ninguna ventaja en absoluto. El modelo mejorado en KeyedCollection se debe casi con certeza al hecho de que fue escrito algunos años después, con el beneficio de la experiencia adquirida. DictionaryBase se considera más útil para garantizar la compatibilidad con versiones anteriores.

ReadOnlyCollection<T>

ReadOnlyCollection<T> es un envoltorio o proxy que proporciona una vista de solo lectura de una colección. Esto es útil para permitir que una clase exponga públicamente acceso de solo lectura a una colección que la clase aún puede actualizar internamente.

Una colección de solo lectura acepta la colección de entrada en su constructor, a la cual mantiene una referencia permanente. No hace una copia estática de la colección de entrada, por lo que los cambios posteriores en la colección de entrada son visibles a través del envoltorio de solo lectura.

Para ilustrar, supongamos que tu clase desea proporcionar acceso público de solo lectura a una lista de cadenas llamada "Names". Podríamos hacerlo de la siguiente manera:

```
public class Test
{
  List<string> names = new List<string>();
  public IReadOnlyList<string> Names => names;
}
```

Aunque "Names" devuelve una interfaz de solo lectura, el consumidor aún puede convertirla en tiempo de ejecución a "List<string>" o "IList<string>" y luego llamar a "Add", "Remove" o "Clear" en la lista. "ReadOnlyCollection<T>" proporciona una solución más robusta:

```
public class Test
{
  List<string> names = new List<string>();
  public ReadOnlyCollection<string> Names { get; private set; }

  public Test() => Names = new ReadOnlyCollection<string> (names);

  public void AddInternally() => names.Add ("test");
}
```

Ahora, solo los miembros dentro de la clase "Test" pueden modificar la lista de nombres:

```
Test t = new Test();

Console.WriteLine (t.Names.Count);       // 0
t.AddInternally();
Console.WriteLine (t.Names.Count);       // 1

t.Names.Add ("test");                    // Compiler error
((IList<string>) t.Names).Add ("test");  // NotSupportedException
```

Colecciones inmutables

Acabamos de describir cómo "ReadOnlyCollection<T>" crea una vista de solo lectura de una colección. Restringir la capacidad de escribir (mutar) una colección, o cualquier otro objeto, simplifica el software y reduce los errores.

Las colecciones inmutables extienden este principio al proporcionar colecciones que no pueden modificarse en absoluto después de la inicialización. Si necesitas agregar un elemento a una colección inmutable, debes instanciar una nueva colección, dejando la antigua sin cambios.

La inmutabilidad es una característica distintiva de la programación funcional y tiene los siguientes beneficios:

- Elimina una gran cantidad de errores asociados con cambios de estado.
- Simplifica en gran medida la paralelización y el multihilo, al prevenir la mayoría de los problemas de seguridad de subprocesos que describimos en los capítulos 14, 22 y 23.
- Facilita el razonamiento sobre el código.

La desventaja de la inmutabilidad es que cuando necesitas realizar un cambio, debes crear un objeto completamente nuevo. Esto conlleva una disminución en el rendimiento, aunque existen estrategias mitigadoras que discutimos en esta sección, incluyendo la capacidad de reutilizar porciones de la estructura original.

Las colecciones inmutables forman parte de .NET (en .NET Framework, están disponibles a través del paquete NuGet "System.Collections.Immutable"). Todas las colecciones están definidas en el espacio de nombres "System.Collections.Immutable":

Type	Internal structure
`ImmutableArray<T>`	Array
`ImmutableList<T>`	AVL tree
`ImmutableDictionary<K,V>`	AVL tree
`ImmutableHashSet<T>`	AVL tree
`ImmutableSortedDictionary<K,V>`	AVL tree
`ImmutableSortedSet<T>`	AVL tree
`ImmutableStack<T>`	Linked list
`ImmutableQueue<T>`	Linked list

Los tipos ImmutableArray<T> e ImmutableList<T> son versiones inmutables de List<T>. Ambos cumplen la misma función, pero con diferentes características de rendimiento que se discuten en "Colecciones inmutables y rendimiento" en la página 387.

Las colecciones inmutables exponen una interfaz pública similar a sus contrapartes mutables. La diferencia clave es que los métodos que parecen alterar la colección (como Add o Remove) no alteran la colección original; en su lugar, devuelven una nueva colección con el elemento solicitado agregado o eliminado.



Las colecciones inmutables evitan agregar y eliminar elementos; no evitan que los propios elementos sean mutados. Para obtener todos los beneficios de la inmutabilidad, debes asegurarte de que solo se agreguen elementos inmutables a una colección inmutable.

Creación de colecciones inmutables

Cada tipo de colección inmutable ofrece un método Create<T>(), que acepta valores iniciales opcionales y devuelve una colección inmutable inicializada:

```
ImmutableArray<int> array = ImmutableArray.Create<int> (1, 2, 3);
```

Cada colección también ofrece un método CreateRange<T>, que hace el mismo trabajo que Create<T>, pero la diferencia es que su tipo de parámetro es IEnumerable<T> en lugar de params T[].

También puedes crear una colección inmutable a partir de un IEnumerable<T> existente utilizando los métodos de extensión apropiados (ToImmutableArray, ToImmutableList, ToImmutableDictionary, etc.):

```
var list = new[] { 1, 2, 3 }.ToImmutableList();
```

Manipulación de colecciones inmutables

El método Add devuelve una nueva colección que contiene los elementos existentes más el nuevo elemento:

```
var oldList = ImmutableList.Create<int> (1, 2, 3);

ImmutableList<int> newList = oldList.Add (4);

Console.WriteLine (oldList.Count);     // 3  (unaltered)
Console.WriteLine (newList.Count);     // 4
```

El método Remove funciona de la misma manera, devolviendo una nueva colección sin el elemento eliminado.

Agregar o eliminar elementos repetidamente de esta manera es ineficiente, ya que se crea una nueva colección inmutable para cada operación de agregar o eliminar. Una solución mejor es llamar a AddRange (o RemoveRange), que acepta un IEnumerable<T> de elementos que se agregan o eliminan de una vez:

```
var anotherList = oldList.AddRange (new[] { 4, 5, 6 });
```

La lista y el array inmutables también definen los métodos Insert e InsertRange para insertar elementos en un índice específico, el método RemoveAt para eliminar en un índice y RemoveAll, que elimina en función de un predicado.

Constructores

Para necesidades de inicialización más complejas, cada clase de colección inmutable define un constructor correspondiente. Los constructores son clases que son funcionalmente equivalentes a una colección mutable, con características de rendimiento similares. Después de inicializar los datos, llamar a `.ToImmutable()` en un constructor devuelve una colección inmutable:

```
ImmutableArray<int>.Builder builder = ImmutableArray.CreateBuilder<int>();
builder.Add (1);
builder.Add (2);
builder.Add (3);
builder.RemoveAt (0);
ImmutableArray<int> myImmutable = builder.ToImmutable();
```

También puedes usar constructores para realizar múltiples actualizaciones en lotes a una colección inmutable existente:

```
var builder2 = myImmutable.ToBuilder();
builder2.Add (4);      // Efficient
builder2.Remove (2);   // Efficient
...                    // More changes to builder...
```

```
// Return a new immutable collection with all the changes applied:
ImmutableArray<int> myImmutable2 = builder2.ToImmutable();
```

Colecciones inmutables y rendimiento

La mayoría de las colecciones inmutables utilizan un árbol AVL internamente, lo que permite reutilizar porciones de la estructura interna original en las operaciones de agregado/eliminación en lugar de tener que recrearla por completo. Esto reduce la sobrecarga de las operaciones de agregado/eliminación, que de otro modo podrían ser potencialmente grandes en colecciones grandes, a algo moderadamente grande, pero a costa de ralentizar las operaciones de lectura. Como resultado, la mayoría de las colecciones inmutables son más lentas que sus contrapartes mutables tanto en lectura como en escritura.

La más afectada es ImmutableList<T>, que tanto en operaciones de lectura como de agregado es de 10 a 200 veces más lenta que List<T> (dependiendo del tamaño de la lista). Por eso existe ImmutableArray<T>: al utilizar un arreglo interno, evita la sobrecarga en las operaciones de lectura (donde su rendimiento es comparable al de un arreglo mutable ordinario). Sin embargo, es mucho más lenta que incluso ImmutableList<T> en operaciones de agregado porque no se puede reutilizar ninguna parte de la estructura original.

Por lo tanto, ImmutableArray<T> es deseable cuando se busca un rendimiento de lectura sin obstáculos y no se esperan muchas llamadas posteriores a Add o Remove (sin usar un constructor):

Type	Read performance	Add performance
ImmutableList<T>	Slow	Slow
ImmutableArray<T>	Very fast	Very slow



Llamar a Remove en un ImmutableArray es más costoso que llamar a Remove en un List<T>, incluso en el peor de los casos al eliminar el primer elemento, porque asignar la nueva colección impone una carga adicional en el recolector de basura.

Aunque las colecciones inmutables en su conjunto incurren en un costo de rendimiento potencialmente significativo, es importante tener en cuenta la magnitud general. Una operación de agregado en un ImmutableList con un millón de elementos es probable que ocurra en menos de un microsegundo en una computadora portátil típica, y una operación de lectura en menos de 100 nanosegundos. Y si necesitas realizar operaciones de escritura en un bucle, puedes evitar el costo acumulado utilizando un constructor.

Los siguientes factores también ayudan a mitigar los costos:

- La inmutabilidad permite una fácil concurrencia y paralelización (Capítulo 23), lo que te permite utilizar todos los núcleos disponibles. La paralelización con estado mutable fácilmente conduce a errores y requiere el uso de bloqueos o colecciones concurrentes, ambos perjudican el rendimiento.

- Con la inmutabilidad, no necesitas realizar "copia defensiva" de colecciones o estructuras de datos para protegerte contra cambios inesperados. Este fue un factor que favoreció el uso de colecciones inmutables en la escritura de partes recientes de Visual Studio.
- En la mayoría de los programas típicos, pocas colecciones tienen la suficiente cantidad de elementos como para que la diferencia importe.

Además de Visual Studio, la cadena de herramientas Microsoft Roslyn, que tiene un buen rendimiento, se construyó con colecciones inmutables, lo que demuestra cómo los beneficios pueden superar a los costos.

Conectando la Igualdad y el Orden

En las secciones "Comparación de Igualdad" en la página 324 y "Comparación de Orden" en la página 335, describimos los protocolos estándar de .NET que hacen que un tipo sea equiparable, hashable y comparable. Un tipo que implementa estos protocolos puede funcionar correctamente en un diccionario o una lista ordenada "tal como está". Más específicamente:

- Un tipo para el cual Equals y GetHashCode devuelven resultados significativos se puede utilizar como clave en un Dictionary o Hashtable.
- Un tipo que implementa IComparable / IComparable<T> se puede utilizar como clave en cualquiera de los diccionarios o listas ordenadas.

La implementación predeterminada de equidad o comparación de un tipo generalmente refleja lo que es más "natural" para ese tipo. Sin embargo, a veces el comportamiento predeterminado no es el deseado. Es posible que necesites un diccionario cuya clave sea de tipo string y se trate sin respetar las mayúsculas y minúsculas. O tal vez desees una lista ordenada de clientes, ordenados por el código postal de cada cliente. Por esta razón, .NET también define un conjunto correspondiente de protocolos "plug-in". Los protocolos "plug-in" logran dos cosas:

- Te permiten cambiar el comportamiento de equidad o comparación de manera alternativa.
- Te permiten usar un diccionario o una colección ordenada con un tipo de clave que no sea intrínsecamente equiparable o comparable.

Los protocolos "plug-in" consisten en las siguientes interfaces:

IEqualityComparer y IEqualityComparer<T>

- Realizan la comparación de igualdad y el hashing "plug-in"
- Reconocidos por Hashtable y Dictionary

IComparer y IComparer<T>

- Realizan la comparación de orden "plug-in"
- Reconocidos por los diccionarios y colecciones ordenadas; también, Array.Sort

Cada interfaz viene en formas genéricas y no genéricas. Las interfaces IEqualityComparer también tienen una implementación predeterminada en una clase llamada EqualityComparer.

Además, existen interfaces llamadas IStructuralEquatable e IStructuralComparable, que permiten la opción de comparaciones estructurales en clases y matrices.

IEqualityComparer y EqualityComparer

Un comparador de igualdad cambia el comportamiento de igualdad y hashing no predeterminado, principalmente para las clases Dictionary y Hashtable.

Recordemos los requisitos de un diccionario basado en tablas hash. Necesita respuestas a dos preguntas para cualquier clave dada:

- ¿Es la misma que otra clave?
- ¿Cuál es su código hash entero?

Un comparador de igualdad responde a estas preguntas implementando las interfaces IEqualityComparer:

```
public interface IEqualityComparer<T>
{
   bool Equals (T x, T y);
   int GetHashCode (T obj);
}

public interface IEqualityComparer     // Nongeneric version
{
   bool Equals (object x, object y);
   int GetHashCode (object obj);
}
```

Para escribir un comparador personalizado, se implementa una o ambas de estas interfaces (implementar ambas proporciona una máxima interoperabilidad). Debido a que esto es algo tedioso, una alternativa es crear una subclase de la clase abstracta EqualityComparer, definida de la siguiente manera:

```
public abstract class EqualityComparer<T> : IEqualityComparer,
                                            IEqualityComparer<T>
{
  public abstract bool Equals (T x, T y);
  public abstract int GetHashCode (T obj);

  bool IEqualityComparer.Equals (object x, object y);
  int IEqualityComparer.GetHashCode (object obj);

  public static EqualityComparer<T> Default { get; }
}
```

EqualityComparer implementa ambas interfaces; tu tarea es simplemente anular los dos métodos abstractos.

La semántica de Equals y GetHashCode sigue las mismas reglas que object.Equals y object.GetHashCode, descritas en el Capítulo 6. En el siguiente ejemplo, definimos una clase Customer con dos campos y luego escribimos un comparador de igualdad que coincide con el primer y último nombre:

```
public class Customer
{
  public string LastName;
  public string FirstName;

  public Customer (string last, string first)
  {
    LastName = last;
    FirstName = first;
  }
}
public class LastFirstEqComparer : EqualityComparer <Customer>
{
  public override bool Equals (Customer x, Customer y)
    => x.LastName == y.LastName && x.FirstName == y.FirstName;

  public override int GetHashCode (Customer obj)
    => (obj.LastName + ";" + obj.FirstName).GetHashCode();
}
```

Para ilustrar cómo funciona, creemos dos clientes:

```
Customer c1 = new Customer ("Bloggs", "Joe");
Customer c2 = new Customer ("Bloggs", "Joe");
```

Como no hemos anulado object.Equals, se aplican las semánticas normales de igualdad de tipos de referencia:

```
Console.WriteLine (c1 == c2);              // False
Console.WriteLine (c1.Equals (c2));        // False
```

Las mismas semánticas de igualdad predeterminadas se aplican al usar estos clientes en un Dictionary sin especificar un comparador de igualdad:

```
var d = new Dictionary<Customer, string>();
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));         // False
```

Ahora, con el comparador de igualdad personalizado:

```
var eqComparer = new LastFirstEqComparer();
var d = new Dictionary<Customer, string> (eqComparer);
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));         // True
```

En este ejemplo, debemos tener cuidado de no cambiar el primer nombre o el apellido del cliente mientras esté en uso en el diccionario; de lo contrario, su código hash cambiaría y el Dictionary se rompería.

EqualityComparer<T>.Default

EqualityComparer<T>.Default devuelve un comparador de igualdad de propósito general que se puede utilizar como alternativa al método estático object.Equals. La ventaja es que primero

verifica si T implementa IEquatable<T>, y si es así, llama a esa implementación evitando el costo de boxing. Esto es particularmente útil en métodos genéricos:

```
static bool Foo<T> (T x, T y)
{
  bool same = EqualityComparer<T>.Default.Equals (x, y);
  ...
```

ReferenceEqualityComparer.Instance (.NET 5+)

A partir de .NET 5, ReferenceEqualityComparer.Instance devuelve un comparador de igualdad que siempre aplica igualdad referencial. En el caso de los tipos de valor, su método Equals siempre devuelve false.

IComparer y Comparer

Los comparadores se utilizan para cambiar la lógica de ordenación personalizada en diccionarios y colecciones ordenadas.

Ten en cuenta que un comparador no es útil para los diccionarios no ordenados como Dictionary y Hashtable, ya que estos requieren un IEqualityComparer para obtener los códigos hash. De manera similar, un comparador de igualdad no es útil para diccionarios y colecciones ordenadas.

Aquí están las definiciones de la interfaz IComparer:

```
public interface IComparer
{
  int Compare(object x, object y);
}
public interface IComparer <in T>
{
  int Compare(T x, T y);
}
```

Al igual que los comparadores de igualdad, existe una clase abstracta que puedes subclasificar en lugar de implementar las interfaces:

```
public abstract class Comparer<T> : IComparer, IComparer<T>
{
   public static Comparer<T> Default { get; }

   public abstract int Compare (T x, T y);       // Implemented by you
   int IComparer.Compare (object x, object y);   // Implemented for you
}
```

El siguiente ejemplo ilustra una clase que describe un deseo, así como un comparador que ordena los deseos por prioridad:

```
class Wish
{
  public string Name;
  public int Priority;

  public Wish (string name, int priority)
  {
    Name = name;
    Priority = priority;
  }
}

class PriorityComparer : Comparer<Wish>
{
  public override int Compare (Wish x, Wish y)
  {
    if (object.Equals (x, y)) return 0;    // Optimization
    if (x == null) return -1;
    if (y == null) return 1;
    return x.Priority.CompareTo (y.Priority);
  }
}
```

La comprobación de object.Equals garantiza que nunca podemos contradecir el método Equals. Llamar al método estático object.Equals en este caso es mejor que llamar a x.Equals porque sigue funcionando si x es nulo.

Así es cómo se utiliza nuestro PriorityComparer para ordenar una lista:

```
var wishList = new List<Wish>();
wishList.Add (new Wish ("Peace", 2));
wishList.Add (new Wish ("Wealth", 3));
wishList.Add (new Wish ("Love", 2));
wishList.Add (new Wish ("3 more wishes", 1));

wishList.Sort (new PriorityComparer());
foreach (Wish w in wishList) Console.Write (w.Name + " | ");

// OUTPUT: 3 more wishes | Love | Peace | Wealth |
```

En el siguiente ejemplo, SurnameComparer te permite ordenar cadenas de apellidos en un orden adecuado para una lista de directorio telefónico:

```
class SurnameComparer : Comparer <string>
{
  string Normalize (string s)
  {
    s = s.Trim().ToUpper();
    if (s.StartsWith ("MC")) s = "MAC" + s.Substring (2);
    return s;
  }

  public override int Compare (string x, string y)
    => Normalize (x).CompareTo (Normalize (y));
}
```

Aquí tienes el uso de SurnameComparer en un diccionario ordenado:

```
var dic = new SortedDictionary<string,string> (new SurnameComparer());
dic.Add ("MacPhail", "second!");
dic.Add ("MacWilliam", "third!");
dic.Add ("McDonald", "first!");

foreach (string s in dic.Values)
  Console.Write (s + " ");              // first! second! third!
```

StringComparer

StringComparer es una clase de complemento predefinida para la equidad y comparación de cadenas, que te permite especificar el idioma y la sensibilidad a mayúsculas y minúsculas. StringComparer implementa tanto IEqualityComparer como IComparer (y sus versiones genéricas), por lo que puedes usarlo con cualquier tipo de diccionario o colección ordenada.

Dado que StringComparer es una clase abstracta, puedes obtener instancias a través de sus propiedades estáticas. StringComparer.Ordinal refleja el comportamiento predeterminado para la comparación de igualdad de cadenas y StringComparer.CurrentCulture para la comparación de orden. Aquí están todos sus miembros estáticos:

```
public static StringComparer CurrentCulture { get; }
public static StringComparer CurrentCultureIgnoreCase { get; }
public static StringComparer InvariantCulture { get; }
public static StringComparer InvariantCultureIgnoreCase { get; }
public static StringComparer Ordinal { get; }
public static StringComparer OrdinalIgnoreCase { get; }
public static StringComparer Create (CultureInfo culture,
                                     bool ignoreCase);
```

En el siguiente ejemplo, se crea un diccionario ordinal y sin distinción entre mayúsculas y minúsculas, de modo que dict["Joe"] y dict["JOE"] significan lo mismo:

```
var dict = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

En el siguiente ejemplo, se ordena una matriz de nombres usando el inglés australiano:

```
string[] names = { "Tom", "HARRY", "sheila" };
CultureInfo ci = new CultureInfo ("en-AU");
Array.Sort<string> (names, StringComparer.Create (ci, false));
```

El último ejemplo es una versión consciente de la cultura de SurnameComparer que escribimos en la sección anterior (para comparar nombres adecuados para una lista de directorio telefónico):

```
class SurnameComparer : Comparer<string>
{
  StringComparer strCmp;

  public SurnameComparer (CultureInfo ci)
  {
    // Create a case-sensitive, culture-sensitive string comparer
    strCmp = StringComparer.Create (ci, false);
  }
```

```
  string Normalize (string s)
  {
    s = s.Trim();
    if (s.ToUpper().StartsWith ("MC")) s = "MAC" + s.Substring (2);
    return s;
  }

  public override int Compare (string x, string y)
  {
    // Directly call Compare on our culture-aware StringComparer
    return strCmp.Compare (Normalize (x), Normalize (y));
  }
}
```

IStructuralEquatable e IStructuralComparable

Como discutimos en el Capítulo 6, las estructuras (structs) implementan la comparación estructural de forma predeterminada: dos estructuras son iguales si todos sus campos son iguales. Sin embargo, a veces la igualdad estructural y la comparación de orden son útiles como opciones adicionales en otros tipos, como los arrays. Las siguientes interfaces ayudan en este sentido:

```
public interface IStructuralEquatable
{
  bool Equals (object other, IEqualityComparer comparer);
  int GetHashCode (IEqualityComparer comparer);
}

public interface IStructuralComparable
{
  int CompareTo (object other, IComparer comparer);
}
```

El IEqualityComparer/IComparer que se pasa como argumento se aplica a cada elemento individual del objeto compuesto. Podemos demostrar esto utilizando arrays. En el siguiente ejemplo, comparamos dos arrays para verificar su igualdad, primero usando el método Equals predeterminado y luego usando la versión de IStructuralEquatable:

```
int[] a1 = { 1, 2, 3 };
int[] a2 = { 1, 2, 3 };
IStructuralEquatable se1 = a1;
Console.Write (a1.Equals (a2));                                    // False
Console.Write (se1.Equals (a2, EqualityComparer<int>.Default));  // True
```

Aquí hay otro ejemplo:

```
string[] a1 = "the quick brown fox".Split();
string[] a2 = "THE QUICK BROWN FOX".Split();
IStructuralEquatable se1 = a1;
bool isTrue = se1.Equals (a2, StringComparer.InvariantCultureIgnoreCase);
```

8 Consultas LINQ

LINQ, o Consulta Integrada en el Lenguaje (por sus siglas en inglés), es un conjunto de características del lenguaje y tiempo de ejecución para escribir consultas estructuradas y seguras en objetos locales y fuentes de datos remotas.

LINQ te permite realizar consultas en cualquier colección que implemente IEnumerable<T>, ya sea un array, una lista o un Document Object Model (DOM) de XML, así como en fuentes de datos remotas como tablas en una base de datos SQL Server. LINQ ofrece los beneficios tanto de la verificación de tipos en tiempo de compilación como de la composición dinámica de consultas.

Este capítulo describe la arquitectura de LINQ y los fundamentos para escribir consultas. Todos los tipos principales están definidos en los espacios de nombres System.Linq y System.Linq.Expressions.


Los ejemplos en este y los dos capítulos siguientes se cargan previamente en una herramienta interactiva de consultas llamada LINQPad. Puedes descargar LINQPad desde http://www.linqpad.net.

Empezando

Las unidades básicas de datos en LINQ son las secuencias (sequences) y los elementos (elements). Una secuencia es cualquier objeto que implementa IEnumerable<T>, y un elemento es cada elemento en la secuencia. En el siguiente ejemplo, names es una secuencia, y "Tom", "Dick" y "Harry" son elementos:

```
string[] names = { "Tom", "Dick", "Harry" };
```

Llamamos a esto una secuencia local porque representa una colección local de objetos en memoria.

Un operador de consulta es un método que transforma una secuencia. Un operador de consulta típico acepta una secuencia de entrada y emite una secuencia de salida transformada. En la clase Enumerable en System.Linq, existen alrededor de 40 operadores de consulta, todos implementados como métodos de extensión estáticos. Estos se llaman operadores de consulta estándar.


Las consultas que operan sobre secuencias locales se llaman consultas locales o consultas LINQ-to-objects.

LINQ también admite secuencias que se pueden alimentar dinámicamente desde una fuente de datos remota, como una base de datos SQL Server. Estas secuencias implementan adicionalmente la interfaz IQueryable<T> y son compatibles a través de un conjunto

correspondiente de operadores de consulta estándar en la clase Queryable. Hablaremos más sobre esto en la sección "Consultas interpretadas" en la página 424.

Una consulta es una expresión que, al ser enumerada, transforma secuencias con operadores de consulta. La consulta más simple consta de una secuencia de entrada y un operador. Por ejemplo, podemos aplicar el operador Where en un simple array para extraer las cadenas cuya longitud es de al menos cuatro caracteres, de la siguiente manera:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = System.Linq.Enumerable.Where
                                    (names, n => n.Length >= 4);
foreach (string n in filteredNames)
  Console.WriteLine (n);

Dick
Harry
```

Debido a que los operadores de consulta estándar están implementados como métodos de extensión, podemos llamar a Where directamente en names, como si fuera un método de instancia:

```
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

Para que esto compile, debes importar el espacio de nombres System.Linq. Aquí tienes un ejemplo completo:

```
using System;
using System.Collections.Generic;
using System.Linq;

string[] names = { "Tom", "Dick", "Harry" };

IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
foreach (string name in filteredNames) Console.WriteLine (name);

Dick
Harry
```



Podríamos acortar aún más nuestro código al utilizar la inferencia de tipos implícita para filteredNames:

```
var filteredNames = names.Where (n => n.Length >= 4);
```

Sin embargo, esto puede dificultar la legibilidad, especialmente fuera de un entorno de desarrollo integrado (IDE), donde no hay sugerencias de herramientas que ayuden. Por esta razón, en este capítulo hacemos un uso menos frecuente de la inferencia de tipos implícita de lo que podrías hacer en tus propios proyectos.

La mayoría de los operadores de consulta aceptan una expresión lambda como argumento. La expresión lambda ayuda a guiar y dar forma a la consulta. En nuestro ejemplo, la expresión lambda es la siguiente:

```
n => n.Length >= 4
```

El argumento de entrada corresponde a un elemento de entrada. En este caso, el argumento de entrada "n" representa cada nombre en el array y es de tipo string. El operador Where requiere que la expresión lambda devuelva un valor bool, que si es verdadero, indica que el elemento debe incluirse en la secuencia de salida. Aquí está su firma:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

La siguiente consulta extrae todos los nombres que contienen la letra "a":

```
IEnumerable<string> filteredNames = names.Where (n => n.Contains ("a"));

foreach (string name in filteredNames)
  Console.WriteLine (name);             // Harry
```

Hasta ahora, hemos construido consultas utilizando métodos de extensión y expresiones lambda. Como verás en breve, esta estrategia es altamente componible, ya que permite encadenar operadores de consulta. En este libro, nos referimos a esto como sintaxis fluida. C# también proporciona otra sintaxis para escribir consultas, llamada sintaxis de expresión de consulta. Aquí está nuestra consulta anterior escrita como una expresión de consulta:

```
IEnumerable<string> filteredNames = from n in names
                                    where n.Contains ("a")
                                    select n;
```

La sintaxis fluida y la sintaxis de expresión de consulta son complementarias. En las dos secciones siguientes, exploramos cada una en más detalle.

Sintaxis Fluida

La sintaxis fluida es la más flexible y fundamental. En esta sección, describimos cómo encadenar operadores de consulta para formar consultas más complejas y mostramos por qué los métodos de extensión son importantes en este proceso. También describimos cómo formular expresiones lambda para un operador de consulta e introducimos varios operadores de consulta nuevos.

Encadenamiento de Operadores de Consulta

En la sección anterior, mostramos dos consultas simples, cada una compuesta por un solo operador de consulta. Para construir consultas más complejas, se agregan operadores de consulta adicionales a la expresión, creando una cadena. Para ilustrar, la siguiente consulta extrae todas las cadenas que contienen la letra "a", las ordena por longitud y luego convierte los resultados a mayúsculas:

```
using System;
using System.Collections.Generic;
using System.Linq;

string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query = names
  .Where   (n => n.Contains ("a"))
  .OrderBy (n => n.Length)
  .Select  (n => n.ToUpper());

foreach (string name in query) Console.WriteLine (name);
```

```
JAY
MARY
HARRY
```



La variable "n" en nuestro ejemplo tiene un ámbito privado dentro de cada una de las expresiones lambda. Podemos reutilizar el identificador "n" por la misma razón que podemos reutilizar el identificador "c" en el siguiente método:

```
void Test()
{
  foreach (char c in "string1") Console.Write (c);
  foreach (char c in "string2") Console.Write (c);
  foreach (char c in "string3") Console.Write (c);
}
```

Where, OrderBy y Select son operadores de consulta estándar que se resuelven como métodos de extensión en la clase Enumerable (si importas el espacio de nombres System.Linq). Ya hemos presentado el operador Where, que emite una versión filtrada de la secuencia de entrada. El operador OrderBy emite una versión ordenada de la secuencia de entrada, y el método Select emite una secuencia en la que cada elemento de entrada se transforma o proyecta con una expresión lambda dada (n.ToUpper(), en este caso). Los datos fluyen de izquierda a derecha a través de la cadena de operadores, por lo que los datos se filtran primero, luego se ordenan y finalmente se proyectan.



Un operador de consulta nunca modifica la secuencia de entrada; en su lugar, devuelve una nueva secuencia. Esto es consistente con el paradigma de programación funcional del que se inspiró LINQ.

Aquí están las firmas de cada uno de estos métodos de extensión (con la firma de OrderBy ligeramente simplificada):

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)

public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)

public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

Cuando los operadores de consulta se encadenan como en este ejemplo, la secuencia de salida de un operador es la secuencia de entrada del siguiente. La consulta completa se asemeja a una línea de producción de cintas transportadoras, como se ilustra en la Figura 8-1.

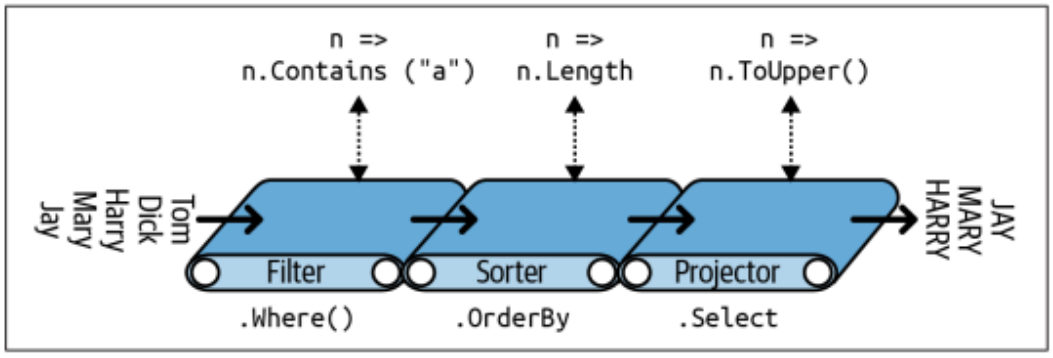


Figure 8-1. Chaining query operators

Podemos construir la misma consulta de forma progresiva de la siguiente manera:

```
// You must import the System.Linq namespace for this to compile:

IEnumerable<string> filtered   = names    .Where   (n => n.Contains ("a"));
IEnumerable<string> sorted     = filtered.OrderBy (n => n.Length);
IEnumerable<string> finalQuery = sorted   .Select  (n => n.ToUpper());
```

`finalQuery` es idéntico a la consulta que construimos anteriormente de forma completa.

Además, cada paso intermedio también es una consulta válida que podemos ejecutar:

```
foreach (string name in filtered)
  Console.Write (name + "|");        // Harry|Mary|Jay|

Console.WriteLine();
foreach (string name in sorted)
  Console.Write (name + "|");        // Jay|Mary|Harry|

Console.WriteLine();
foreach (string name in finalQuery)
  Console.Write (name + "|");        // JAY|MARY|HARRY|
```

Por qué son importantes los métodos de extensión

En lugar de utilizar la sintaxis de métodos de extensión, puedes utilizar la sintaxis convencional de métodos estáticos para llamar a los operadores de consulta:

```
IEnumerable<string> filtered = Enumerable.Where (names,
                                                 n => n.Contains ("a"));
IEnumerable<string> sorted = Enumerable.OrderBy (filtered, n => n.Length);
IEnumerable<string> finalQuery = Enumerable.Select (sorted,
                                                    n => n.ToUpper());
```

De hecho, así es como el compilador traduce las llamadas a los métodos de extensión. Sin embargo, renunciar a los métodos de extensión tiene un costo si deseas escribir una consulta en una sola instrucción, como hicimos anteriormente. Veamos nuevamente la consulta en una sola instrucción, pero esta vez utilizando la sintaxis de métodos de extensión:

```
IEnumerable<string> query = names.Where   (n => n.Contains ("a"))
                                  .OrderBy (n => n.Length)
                                  .Select  (n => n.ToUpper());
```

Su forma lineal y natural refleja el flujo de datos de izquierda a derecha y también mantiene las expresiones lambda junto a sus operadores de consulta (notación infija). Sin los métodos de extensión, la consulta pierde su fluidez:

```
IEnumerable<string> query =
  Enumerable.Select (
    Enumerable.OrderBy (
      Enumerable.Where (
        names, n => n.Contains ("a")
      ), n => n.Length
    ), n => n.ToUpper()
  );
```

Componiendo expresiones lambda

En ejemplos anteriores, pasamos la siguiente expresión lambda al operador Where:

```
n => n.Contains ("a")      // Input type = string, return type = bool.
```



Una expresión lambda que toma un valor y devuelve un bool se llama predicado.

El propósito de la expresión lambda depende del operador de consulta en particular. Con el operador Where, indica si un elemento debe incluirse en la secuencia de salida. En el caso del operador OrderBy, la expresión lambda asigna a cada elemento de la secuencia de entrada su clave de ordenamiento. Con el operador Select, la expresión lambda determina cómo se transforma cada elemento de la secuencia de entrada antes de ser alimentado a la secuencia de salida.



Una expresión lambda en un operador de consulta siempre trabaja con elementos individuales de la secuencia de entrada, no con la secuencia en su conjunto.

El operador de consulta evalúa su expresión lambda bajo demanda, típicamente una vez por elemento de la secuencia de entrada. Las expresiones lambda te permiten introducir tu propia lógica en los operadores de consulta. Esto hace que los operadores de consulta sean versátiles y, al mismo tiempo, simples en su implementación interna. Aquí tienes una implementación completa de Enumerable.Where, dejando de lado el manejo de excepciones:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

Las expresiones lambda y las firmas de Func

Los operadores de consulta estándar utilizan delegados genéricos Func. Func es una familia de delegados genéricos de propósito general en el espacio de nombres System, definidos con la siguiente intención:

> Los argumentos de tipo en Func aparecen en el mismo orden que en las expresiones lambda.

Por lo tanto, Func<TSource, bool> se corresponde con una expresión lambda TSource=>bool: una que acepta un argumento TSource y devuelve un valor bool.

De manera similar, Func<TSource, TResult> se corresponde con una expresión lambda TSource=>TResult.

Los delegados Func se enumeran en "Expresiones lambda" en la página 174.

Expresiones lambda y tipificación de elementos

Los operadores de consulta estándar utilizan los siguientes nombres de parámetros de tipo:

Generic type letter	Meaning
TSource	Element type for the input sequence
TResult	Element type for the output sequence (if different from TSource)
TKey	Element type for the *key* used in sorting, grouping, or joining

TSource se determina según la secuencia de entrada. TResult y TKey se infieren generalmente a partir de la expresión lambda que proporcionas.

Por ejemplo, considera la firma del operador de consulta Select:

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

Func<TSource, TResult> se corresponde con una expresión lambda TSource=>TResult: una que asigna un elemento de entrada a un elemento de salida. TSource y TResult pueden ser tipos diferentes, por lo que la expresión lambda puede cambiar el tipo de cada elemento. Además, la

expresión lambda determina el tipo de secuencia de salida. La siguiente consulta utiliza Select para transformar elementos de tipo string en elementos de tipo entero:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<int> query = names.Select (n => n.Length);

foreach (int length in query)
  Console.Write (length + "|");    // 3|4|5|4|3|
```

El compilador puede inferir el tipo de TResult a partir del valor de retorno de la expresión lambda. En este caso, n.Length devuelve un valor int, por lo que se infiere que TResult es int.

El operador de consulta Where es más simple y no requiere inferencia de tipos para la salida porque los elementos de entrada y salida son del mismo tipo. Esto tiene sentido porque el operador simplemente filtra elementos; no los transforma:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

Finalmente, considera la firma del operador de consulta OrderBy:

```
// Slightly simplified:
public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)
```

Func<TSource,TKey> asigna un elemento de entrada a una clave de ordenación. TKey se infiere a partir de la expresión lambda y es independiente de los tipos de elemento de entrada y salida. Por ejemplo, podríamos elegir ordenar una lista de nombres por longitud (clave int) o alfabéticamente (clave string):

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> sortedByLength, sortedAlphabetically;
sortedByLength       = names.OrderBy (n => n.Length);   // int key
sortedAlphabetically = names.OrderBy (n => n);          // string key
```



Puedes llamar a los operadores de consulta en Enumerable con delegados tradicionales que se refieren a métodos en lugar de expresiones lambda. Este enfoque es efectivo para simplificar ciertos tipos de consultas locales, especialmente con LINQ to XML, y se demuestra en el Capítulo 10. Sin embargo, no funciona con secuencias basadas en IQueryable<T> (por ejemplo, al consultar una base de datos) porque los operadores en Queryable requieren expresiones lambda para emitir árboles de expresión. Discutiremos esto más adelante en "Consultas interpretadas" en la página 424.

Orden natural

El orden original de los elementos dentro de una secuencia de entrada es importante en LINQ. Algunos operadores de consulta dependen de este orden, como Take, Skip y Reverse.

El operador Take devuelve los primeros x elementos, descartando el resto:

```
int[] numbers  = { 10, 9, 8, 7, 6 };
IEnumerable<int> firstThree = numbers.Take (3);     // { 10, 9, 8 }
```

El operador Skip ignora los primeros x elementos y devuelve el resto:

```
IEnumerable<int> lastTwo    = numbers.Skip (3);     // { 7, 6 }
```

Reverse hace exactamente lo que dice:

```
IEnumerable<int> reversed   = numbers.Reverse();    // { 6, 7, 8, 9, 10 }
```

Con consultas locales (LINQ-to-objects), los operadores como Where y Select preservan el orden original de la secuencia de entrada (al igual que todos los demás operadores de consulta, excepto aquellos que cambian específicamente el orden).

Otros operadores

No todos los operadores de consulta devuelven una secuencia. Los operadores de elemento extraen un elemento de la secuencia de entrada; ejemplos son First, Last y ElementAt:

```
int[] numbers    = { 10, 9, 8, 7, 6 };
int firstNumber  = numbers.First();                       // 10
int lastNumber   = numbers.Last();                        // 6
int secondNumber = numbers.ElementAt(1);                  // 9
int secondLowest = numbers.OrderBy(n=>n).Skip(1).First(); // 7
```

Dado que estos operadores devuelven un solo elemento, normalmente no llamas a otros operadores de consulta en su resultado a menos que ese elemento en sí sea una colección.

Los operadores de agregación devuelven un valor escalar, generalmente de tipo numérico:

```
int count = numbers.Count();          // 5;
int min = numbers.Min();              // 6;
```

Los cuantificadores devuelven un valor bool:

```
bool hasTheNumberNine = numbers.Contains (9);          // true
bool hasMoreThanZeroElements = numbers.Any();          // true
bool hasAnOddElement = numbers.Any (n => n % 2 != 0);  // true
```

Algunos operadores de consulta aceptan dos secuencias de entrada. Ejemplos son Concat, que une una secuencia a otra, y Union, que hace lo mismo pero eliminando duplicados:

```
int[] seq1 = { 1, 2, 3 };
int[] seq2 = { 3, 4, 5 };
IEnumerable<int> concat = seq1.Concat (seq2);    //  { 1, 2, 3, 3, 4, 5 }
IEnumerable<int> union  = seq1.Union (seq2);     //  { 1, 2, 3, 4, 5 }
```

Los operadores de unión también entran en esta categoría. El Capítulo 9 cubre todos los operadores de consulta en detalle.

Expresiones de consulta

C# proporciona un atajo sintáctico para escribir consultas LINQ, llamado expresiones de consulta. Contrariamente a la creencia popular, una expresión de consulta no es una forma de incrustar SQL en C#. De hecho, el diseño de las expresiones de consulta se inspiró principalmente en las comprensiones de listas de lenguajes de programación funcionales como LISP y Haskell, aunque el SQL tuvo una influencia cosmética.



En este libro, nos referimos a la sintaxis de expresiones de consulta simplemente como sintaxis de consulta.

En la sección anterior, escribimos una consulta con sintaxis fluida para extraer cadenas que contienen la letra "a", ordenadas por longitud y convertidas a mayúsculas. Aquí está el mismo ejemplo en sintaxis de consulta:

```
using System;
using System.Collections.Generic;
using System.Linq;

string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query =
  from    n in names
  where   n.Contains ("a")     // Filter elements
  orderby n.Length             // Sort elements
  select  n.ToUpper();         // Translate each element (project)

foreach (string name in query) Console.WriteLine (name);

JAY
MARY
HARRY
```

Las expresiones de consulta siempre comienzan con una cláusula `from` y terminan con una cláusula `select` o `group`. La cláusula `from` declara una variable de rango (en este caso, `n`), que puedes pensar como un recorrido de la secuencia de entrada, similar a un `foreach`. La Figura 8-2 ilustra la sintaxis completa en forma de diagrama de ferrocarril.



Para leer este diagrama, comienza en la parte izquierda y luego continúa a lo largo de la vía como si fueras un tren. Por ejemplo, después de la cláusula `from` obligatoria, puedes incluir opcionalmente una cláusula `orderby`, `where`, `let` o `join`. Después de eso, puedes continuar con una cláusula `select` o `group`, o retroceder e incluir otra cláusula `from`, `orderby`, `where`, `let` o `join`.

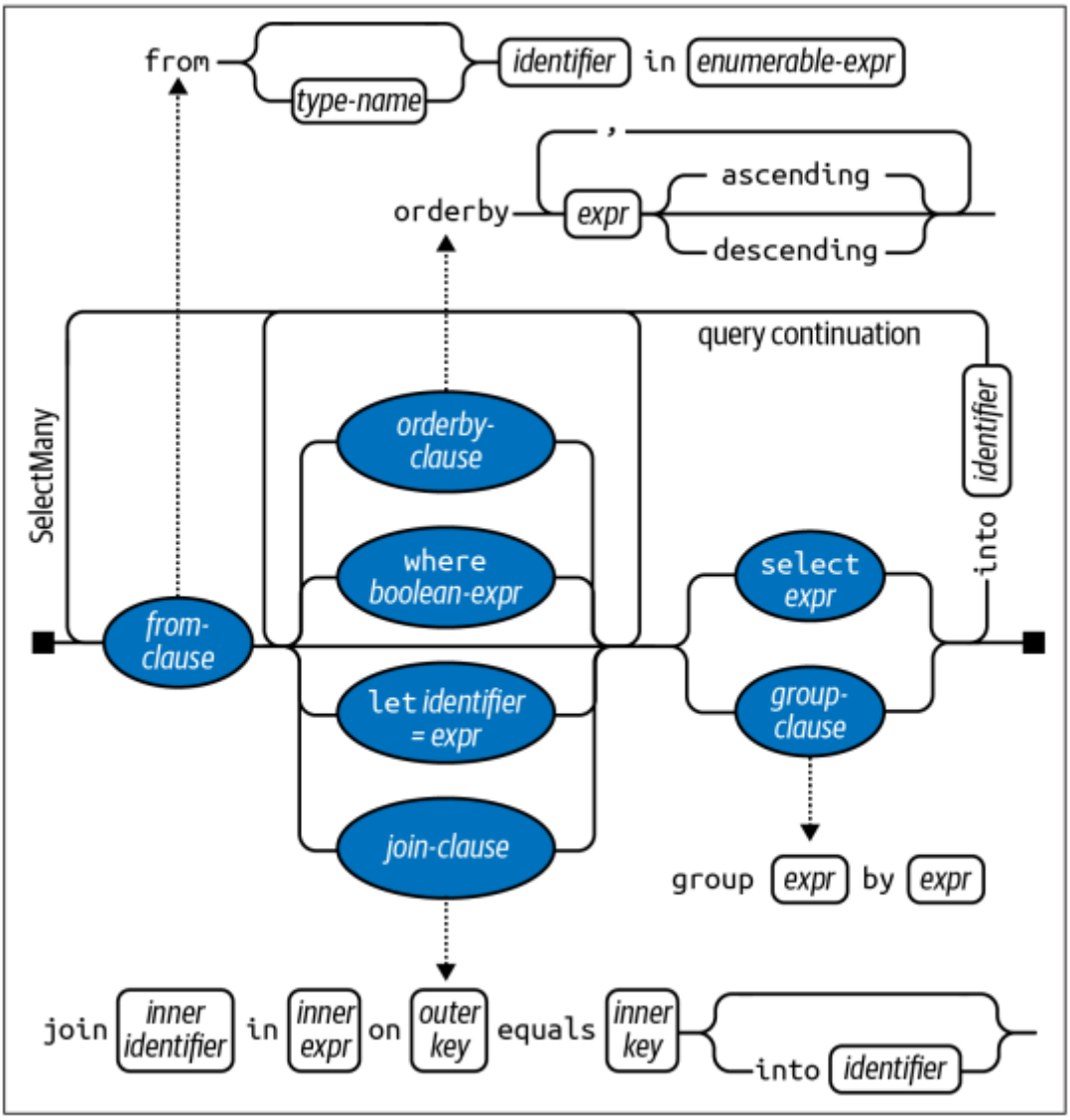


Figure 8-2. Query syntax

El compilador procesa una expresión de consulta traduciéndola en una sintaxis fluida. Lo hace de manera bastante mecánica, al igual que traduce las declaraciones foreach en llamadas a GetEnumerator y MoveNext. Esto significa que todo lo que puedes escribir en sintaxis de consulta también puedes escribirlo en sintaxis fluida. El compilador traduce inicialmente nuestra consulta de ejemplo en lo siguiente:

```
IEnumerable<string> query = names.Where   (n => n.Contains ("a"))
                                 .OrderBy (n => n.Length)
                                 .Select  (n => n.ToUpper());
```

Los operadores Where, OrderBy y Select se resuelven utilizando las mismas reglas que se aplicarían si la consulta se escribiera en sintaxis fluida. En este caso, se enlazan a métodos de extensión en la clase Enumerable porque se importa el espacio de nombres System.Linq y `names` implementa `IEnumerable<string>`. Sin embargo, el compilador no favorece específicamente la clase Enumerable al traducir expresiones de consulta. Puedes pensar en el compilador como si insertara mecánicamente las palabras "Where", "OrderBy" y "Select" en la declaración y luego lo compilara como si hubieras escrito los nombres de los métodos tú mismo.

Esto ofrece flexibilidad en cómo se resuelven. Por ejemplo, los operadores en las consultas de base de datos que escribimos en secciones posteriores se enlazarán a métodos de extensión en Queryable.



Si eliminamos la directiva using System.Linq de nuestro programa, la consulta no se compilaría, ya que los métodos Where, OrderBy y Select no tendrían dónde enlazarse. Las expresiones de consulta no pueden compilar a menos que importes System.Linq o cualquier otro espacio de nombres con una implementación de estos métodos de consulta.

Variables de rango

El identificador que sigue inmediatamente la palabra clave from se llama variable de rango. Una variable de rango se refiere al elemento actual en la secuencia sobre la cual se realizará la operación.

En nuestros ejemplos, la variable de rango n aparece en cada cláusula de la consulta. Y sin embargo, la variable en realidad enumera sobre una secuencia diferente en cada cláusula:

```
from    n in names          // n is our range variable
where   n.Contains ("a")    // n = directly from the array
orderby n.Length            // n = subsequent to being filtered
select  n.ToUpper()         // n = subsequent to being sorted
```

Esto se vuelve claro cuando examinamos la traducción mecánica del compilador a sintaxis fluida:

```
names.Where   (n => n.Contains ("a"))      // Locally scoped n
     .OrderBy (n => n.Length)              // Locally scoped n
     .Select  (n => n.ToUpper())           // Locally scoped n
```

Como puedes ver, cada instancia de n tiene un alcance privado en su propia expresión lambda.

Las expresiones de consulta también te permiten introducir nuevas variables de rango a través de las siguientes cláusulas:

- let
- into
- Una cláusula adicional de from
- join

Estos se cubren más adelante en este capítulo en "Estrategias de composición" en la página 418, así como en el Capítulo 9, en "Proyección" en la página 449 y "Unión" en la página 449.

Sintaxis de consulta versus sintaxis SQL

Las expresiones de consulta se parecen superficialmente a SQL, pero son muy diferentes. Una consulta de LINQ se reduce a una expresión de C#, por lo que sigue las reglas estándar de C#. Por ejemplo, con LINQ no puedes usar una variable antes de declararla. En SQL, puedes hacer referencia a un alias de tabla en la cláusula SELECT antes de definirlo en una cláusula FROM. Una

subconsulta en LINQ es simplemente otra expresión de C# y, por lo tanto, no requiere una sintaxis especial. Las subconsultas en SQL están sujetas a reglas especiales.

Con LINQ, los datos fluyen lógicamente de izquierda a derecha a través de la consulta. Con SQL, el orden es menos estructurado en cuanto al flujo de datos.

Una consulta de LINQ consta de una cinta transportadora o canalización de operadores que aceptan y emiten secuencias cuyo orden de elementos puede ser importante. Una consulta de SQL consta de una red de cláusulas que trabajan principalmente con conjuntos desordenados.

La sintaxis de consulta y la sintaxis fluida tienen ventajas distintas.

La sintaxis de consulta es más sencilla para consultas que involucran cualquiera de los siguientes puntos:

- Una cláusula let para introducir una nueva variable junto con la variable de rango.
- SelectMany, Join o GroupJoin, seguidos de una referencia externa a la variable de rango.

(Describimos la cláusula let en "Estrategias de composición" en la página 418; describimos SelectMany, Join y GroupJoin en el Capítulo 9).

El punto intermedio son las consultas que involucran el uso simple de Where, OrderBy y Select. Ambas sintaxis funcionan bien; la elección en este caso es en gran medida personal.

Para consultas que constan de un solo operador, la sintaxis fluida es más corta y menos desordenada.

Finalmente, hay muchos operadores que no tienen una palabra clave en la sintaxis de consulta. Estos requieren que uses la sintaxis fluida, al menos en parte. Esto significa cualquier operador fuera de los siguientes:

```
Where, Select, SelectMany
OrderBy, ThenBy, OrderByDescending, ThenByDescending
GroupBy, Join, GroupJoin
```

Consultas de sintaxis mixta

Si un operador de consulta no tiene soporte de sintaxis de consulta, puedes combinar la sintaxis de consulta y la sintaxis fluida. La única restricción es que cada componente de sintaxis de consulta debe estar completo (es decir, comenzar con una cláusula from y terminar con una cláusula select o group).

Suponiendo esta declaración de arreglo:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

El siguiente ejemplo cuenta el número de nombres que contienen la letra "a":

```
int matches = (from n in names where n.Contains ("a") select n).Count();
// 3
```

La siguiente consulta obtiene el primer nombre en orden alfabético:

```
string first = (from n in names orderby n select n).First();   // Dick
```

El enfoque de sintaxis mixta a veces es beneficioso en consultas más complejas. Sin embargo, con estos ejemplos simples, podríamos quedarnos con la sintaxis fluida en todo momento sin penalización:

```
int matches = names.Where (n => n.Contains ("a")).Count();   // 3
string first = names.OrderBy (n => n).First();               // Dick
```



Hay momentos en los que las consultas de sintaxis mixta ofrecen una gran funcionalidad y simplicidad. Es importante no favorecer unilateralmente la sintaxis de consulta o fluida; de lo contrario, no podrás escribir consultas de sintaxis mixta cuando sean la mejor opción.

Cuando corresponda, el resto de este capítulo muestra conceptos clave tanto en sintaxis fluida como en sintaxis de consulta.

Ejecución Diferida

Una característica importante de la mayoría de los operadores de consulta es que se ejecutan no cuando se construyen, sino cuando se enumeran (es decir, cuando se llama a MoveNext en su enumerador). Considera la siguiente consulta:

```
var numbers = new List<int> { 1 };

IEnumerable<int> query = numbers.Select (n => n * 10);    // Build query

numbers.Add (2);                    // Sneak in an extra element

foreach (int n in query)
  Console.Write (n + "|");          // 10|20|
```

El número adicional que agregamos a la lista después de construir la consulta se incluye en el resultado porque no es hasta que se ejecuta la instrucción foreach que se realiza cualquier filtrado o ordenamiento. Esto se conoce como ejecución diferida o perezosa y es lo mismo que ocurre con los delegados:

```
Action a = () => Console.WriteLine ("Foo");
// We've not written anything to the Console yet. Now let's run it:
a();  // Deferred execution!
```

Todos los operadores de consulta estándar proporcionan ejecución diferida, con las siguientes excepciones:

- Operadores que devuelven un único elemento o un valor escalar, como First o Count
- Los siguientes operadores de conversión.

```
ToArray, ToList, ToDictionary, ToLookup, ToHashSet
```

Estos operadores causan una ejecución inmediata de la consulta porque sus tipos de resultado no tienen un mecanismo para proporcionar ejecución diferida. El método Count, por ejemplo, devuelve un entero simple, que luego no se enumera. La siguiente consulta se ejecuta de inmediato:

```
int matches = numbers.Where (n => n <= 2).Count();    // 1
```

La ejecución diferida es importante porque desacopla la construcción de la consulta de la ejecución de la consulta. Esto te permite construir una consulta en varios pasos y también hace posibles las consultas a bases de datos.



Las subconsultas proporcionan otro nivel de indirección. Todo en una subconsulta está sujeto a ejecución diferida, incluyendo los métodos de agregación y conversión. Esto se describe en "Subqueries" en la página 414.

Reevaluación

La ejecución diferida tiene otra consecuencia: una consulta con ejecución diferida se reevalúa cuando se vuelve a enumerar:

```
var numbers = new List<int>() { 1, 2 };

IEnumerable<int> query = numbers.Select (n => n * 10);
foreach (int n in query) Console.Write (n + "|");   // 10|20|

numbers.Clear();
foreach (int n in query) Console.Write (n + "|");   // <nothing>
```

Hay un par de razones por las que la reevaluación a veces es desventajosa:

- A veces, quieres "congelar" o almacenar en caché los resultados en un cierto punto en el tiempo.
- Algunas consultas son computacionalmente intensivas (o dependen de consultas a una base de datos remota), por lo que no quieres repetirlas innecesariamente.

Puedes evitar la reevaluación llamando a un operador de conversión como ToArray o ToList. ToArray copia la salida de una consulta a un arreglo; ToList copia a una lista genérica List<T>:

```
var numbers = new List<int>() { 1, 2 };

List<int> timesTen = numbers
  .Select (n => n * 10)

  .ToList();               // Executes immediately into a List<int>

numbers.Clear();
Console.WriteLine (timesTen.Count);      // Still 2
```

Captured Variables

Si las expresiones lambda de tu consulta capturan variables externas, la consulta respetará el valor de esas variables en el momento en que se ejecuta la consulta:

```
int[] numbers = { 1, 2 };

int factor = 10;
IEnumerable<int> query = numbers.Select (n => n * factor);
factor = 20;
foreach (int n in query) Console.Write (n + "|");   // 20|40|
```

Esto puede ser una trampa al construir una consulta dentro de un bucle for. Por ejemplo, supongamos que queremos eliminar todas las vocales de una cadena. Lo siguiente, aunque ineficiente, da el resultado correcto:

```
IEnumerable<char> query = "Not what you might expect";

query = query.Where (c => c != 'a');
query = query.Where (c => c != 'e');
query = query.Where (c => c != 'i');
query = query.Where (c => c != 'o');
query = query.Where (c => c != 'u');

foreach (char c in query) Console.Write (c);  // Nt wht y mght xpct
```

Ahora observa lo que sucede cuando refactorizamos esto con un bucle for:

```
IEnumerable<char> query = "Not what you might expect";
string vowels = "aeiou";

for (int i = 0; i < vowels.Length; i++)
  query = query.Where (c => c != vowels[i]);

foreach (char c in query) Console.Write (c);
```

Se lanza una excepción IndexOutOfRangeException al enumerar la consulta porque, como vimos en el Capítulo 4 (ver "Capturing Outer Variables" en la página 176), el compilador asigna ámbito a la variable de iteración en el bucle for como si estuviera declarada fuera del bucle. Por lo tanto, cada cierre captura la misma variable (i) cuyo valor es 5 cuando se enumera la consulta en realidad. Para resolver esto, debes asignar la variable del bucle a otra variable declarada dentro del bloque de instrucciones:

```
for (int i = 0; i < vowels.Length; i++)
{
  char vowel = vowels[i];
  query = query.Where (c => c != vowel);
}
```

Esto obliga a que se capture una nueva variable local en cada iteración del bucle.



Otra forma de resolver el problema es reemplazar el bucle for por un bucle foreach:

```
foreach (char vowel in vowels)
  query = query.Where (c => c != vowel);
```

Cómo funciona la ejecución aplazada

Los operadores de consulta proporcionan ejecución aplazada mediante la devolución de secuencias decoradoras. A diferencia de una clase de colección tradicional como un array o una lista enlazada, una secuencia decoradora (en general) no tiene una estructura interna propia para almacenar elementos. En su lugar, envuelve otra secuencia que se suministra en tiempo de ejecución, a la cual mantiene una dependencia permanente. Cada vez que solicitas datos de una decoradora, esta a su vez debe solicitar datos a la secuencia de entrada envuelta.



La transformación del operador de consulta constituye la "decoración". Si la secuencia de salida no realizara ninguna transformación, sería un proxy en lugar de una decoradora.

Llamar a Where simplemente construye la secuencia decoradora envolvente, que mantiene una referencia a la secuencia de entrada, la expresión lambda y cualquier otro argumento suministrado. La secuencia de entrada solo se enumera cuando se enumera la decoradora.

La Figura 8-3 ilustra la composición de la siguiente consulta:

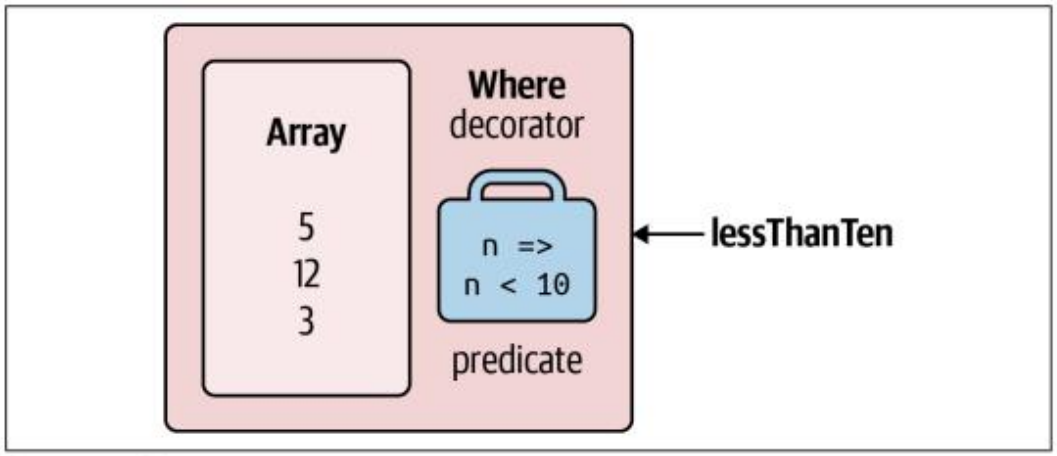


Figure 8-3. Decorator sequence

Cuando enumeras `lessThanTen`, estás, en efecto, consultando el array a través de la decoradora Where.

La buena noticia, en caso de que alguna vez desees escribir tu propio operador de consulta, es que implementar una secuencia decoradora es fácil con un iterador en C#. Aquí te muestro cómo puedes escribir tu propio método Select:

```
public static IEnumerable<TResult> MySelect<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

Este método es un iterador gracias a la declaración `yield return`. Funcionalmente, es un atajo para el siguiente código:

```
public static IEnumerable<TResult> MySelect<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  return new SelectSequence (source, selector);
}
```

donde `SelectSequence` es una clase (generada por el compilador) cuyo enumerador encapsula la lógica del método iterador.

Por lo tanto, cuando llamas a un operador como Select o Where, no estás haciendo más que instanciar una clase enumerable que decora la secuencia de entrada.

Encadenamiento de decoradores

El encadenamiento de operadores de consulta crea una superposición de decoradores. Considera la siguiente consulta:

```
IEnumerable<int> query = new int[] { 5, 12, 3 }.Where   (n => n < 10)
                                                .OrderBy (n => n)
                                                .Select  (n => n * 10);
```

Cada operador de consulta instancia un nuevo decorador que envuelve la secuencia anterior (similar a una muñeca rusa). La Figura 8-4 ilustra el modelo de objetos de esta consulta. Es importante destacar que este modelo de objetos se construye por completo antes de cualquier enumeración.

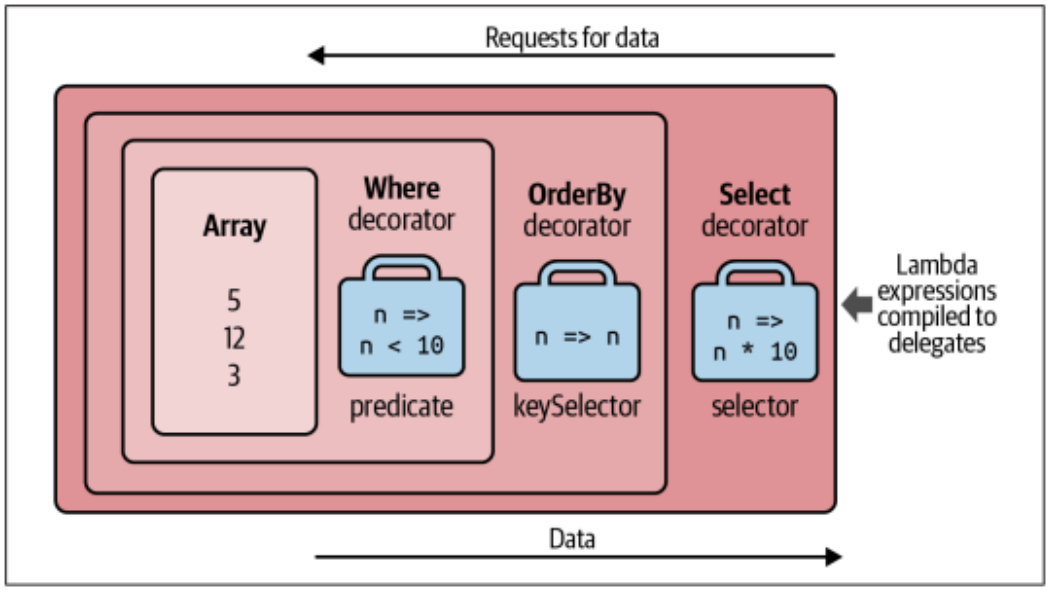


Figure 8-4. Layered decorator sequences

Cuando enumeras `query`, estás consultando el arreglo original transformado a través de una serie de decoradores o encadenamientos.



Agregar `ToList` al final de esta consulta haría que los operadores anteriores se ejecuten de inmediato, colapsando todo el modelo de objetos en una sola lista.

La Figura 8-5 muestra la misma composición de objetos en la sintaxis de Unified Modeling Language (UML). El decorador de `Select` hace referencia al decorador de `OrderBy`, que a su vez hace referencia al decorador de `Where`, que a su vez hace referencia al arreglo. Una característica de la ejecución diferida es que se construye el mismo modelo de objetos si compones la consulta de forma progresiva:

```
IEnumerable<int>
  source    = new int[] { 5, 12, 3 },
  filtered  = source   .Where   (n => n < 10),
  sorted    = filtered .OrderBy (n => n),
  query     = sorted   .Select  (n => n * 10);
```

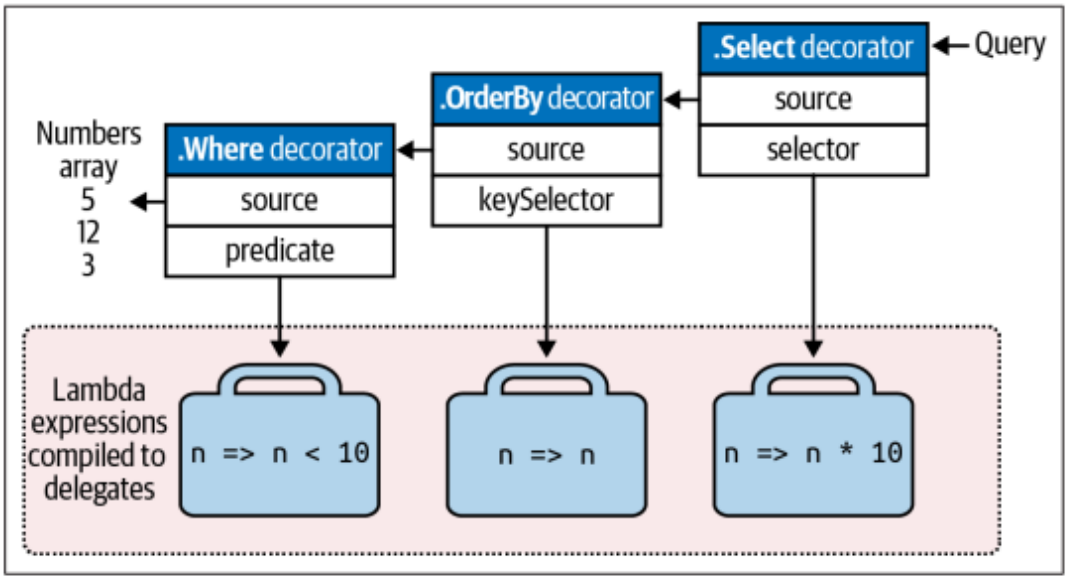


Figure 8-5. UML decorator composition

Cómo las consultas son ejecutadas

Aquí están los resultados de la enumeración de la consulta anterior:

```
foreach (int n in query) Console.WriteLine (n);

30
50
```

Detrás de escena, el bucle `foreach` llama a `GetEnumerator` en el decorador de `Select` (el último u externo operador), lo cual inicia todo el proceso. El resultado es una cadena de enumeradores que refleja estructuralmente la cadena de secuencias de decoradores. La Figura 8-6 ilustra el flujo de ejecución a medida que avanza la enumeración.

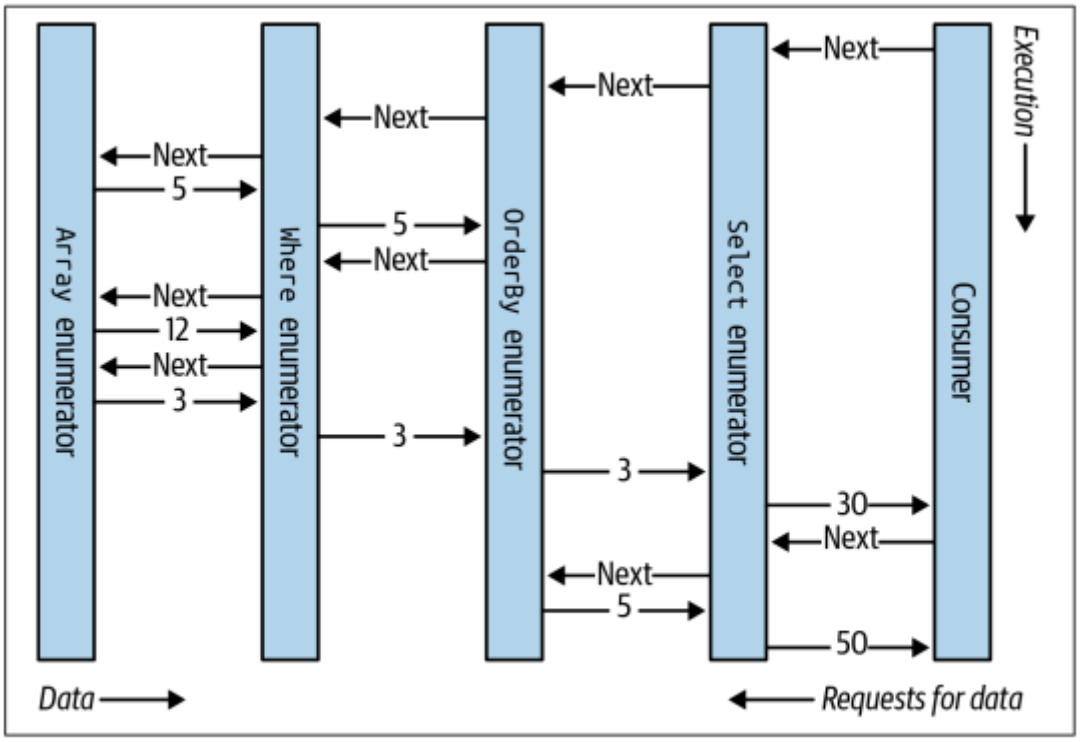


Figure 8-6. Execution of a local query

En la primera sección de este capítulo, representamos una consulta como una línea de producción de cintas transportadoras. Ampliando esta analogía, podemos decir que una consulta LINQ es una línea de producción perezosa, donde las cintas transportadoras solo se mueven cuando se solicita. Construir una consulta es construir una línea de producción, con todo en su lugar, pero sin movimiento. Luego, cuando el consumidor solicita un elemento (enumera la consulta), la cinta transportadora más a la derecha se activa, lo que a su vez activa a las demás cintas transportadoras a medida que se necesitan elementos de la secuencia de entrada. LINQ sigue un modelo de extracción impulsado por la demanda, en lugar de un modelo de empuje impulsado por el suministro. Esto es importante, como veremos más adelante, para permitir que LINQ se ajuste a consultas en bases de datos SQL.

Subqueries

Una subconsulta es una consulta contenida dentro de la expresión lambda de otra consulta. El siguiente ejemplo utiliza una subconsulta para ordenar músicos por su apellido:

```
string[] musos =
  { "David Gilmour", "Roger Waters", "Rick Wright", "Nick Mason" };

IEnumerable<string> query = musos.OrderBy (m => m.Split().Last());
```

`m.Split()` convierte cada cadena en una colección de palabras, sobre la cual luego llamamos al operador de consulta `Last`. `m.Split().Last` es la subconsulta; `query` hace referencia a la consulta externa.

Las subconsultas son permitidas porque puedes poner cualquier expresión válida de C# en el lado derecho de una expresión lambda. Una subconsulta es simplemente otra expresión de C#.

Esto significa que las reglas para las subconsultas son una consecuencia de las reglas para las expresiones lambda (y el comportamiento de los operadores de consulta en general).



El término "subconsulta", en sentido general, tiene un significado más amplio. Para describir LINQ, utilizamos el término solo para una consulta referenciada desde dentro de la expresión lambda de otra consulta. En una expresión de consulta, una subconsulta se refiere a una consulta desde una expresión en cualquier cláusula excepto la cláusula from.

Una subconsulta tiene un ámbito privado para la expresión que la contiene y puede hacer referencia a parámetros en la expresión lambda externa (o variables de rango en una expresión de consulta).

`m.Split().Last` es una subconsulta muy simple. La siguiente consulta recupera todas las cadenas en un arreglo cuya longitud coincide con la longitud de la cadena más corta:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> outerQuery = names
  .Where (n => n.Length == names.OrderBy (n2 => n2.Length)
                               .Select  (n2 => n2.Length).First());

// Tom, Jay
```

Aquí está lo mismo como una expresión de consulta:

```
IEnumerable<string> outerQuery =
  from   n in names
  where  n.Length ==
           (from n2 in names orderby n2.Length select n2.Length).First()
  select n;
```

Debido a que la variable de rango externa (n) está en ámbito para una subconsulta, no podemos reutilizar n como la variable de rango de la subconsulta.

Una subconsulta se ejecuta cada vez que se evalúa la expresión lambda que la contiene. Esto significa que una subconsulta se ejecuta bajo demanda, a discreción de la consulta externa. Podríamos decir que la ejecución se realiza de afuera hacia adentro. Las consultas locales siguen este modelo literalmente; las consultas interpretadas (por ejemplo, consultas a bases de datos) siguen este modelo conceptualmente.

La subconsulta se ejecuta según sea necesario para alimentar la consulta externa. Como se ilustra en las Figuras 8-7 y 8-8, la subconsulta en nuestro ejemplo (la cinta transportadora superior en la Figura 8-7) se ejecuta una vez por cada iteración del bucle externo.

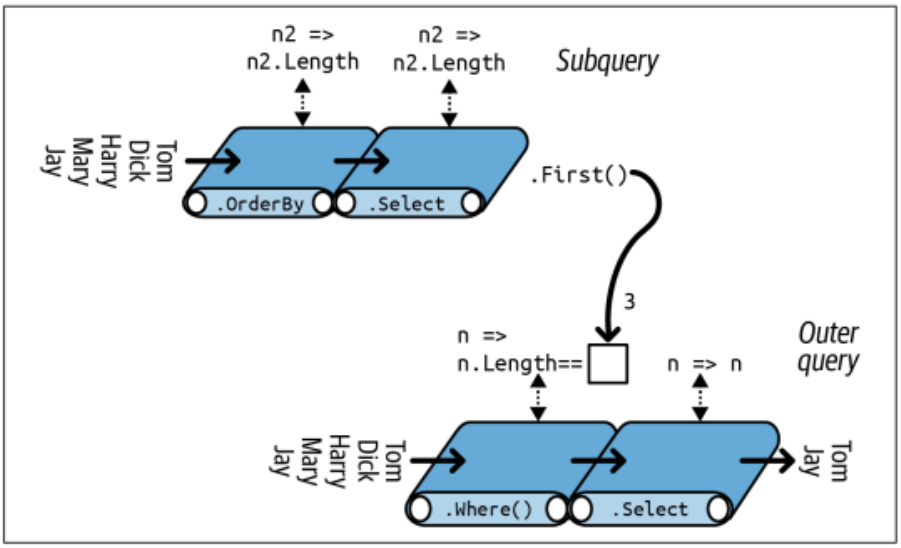


Figure 8-7. Subquery composition

Podemos expresar nuestra subconsulta anterior de manera más concisa de la siguiente manera:

```
IEnumerable<string> query =
  from   n in names
  where  n.Length == names.OrderBy (n2 => n2.Length).First().Length
  select n;
```

Con la función de agregación Min, podemos simplificar aún más la consulta:

```
IEnumerable<string> query =
  from   n in names
  where  n.Length == names.Min (n2 => n2.Length)
  select n;
```

En "Consultas interpretadas" en la página 424, describimos cómo se pueden consultar fuentes remotas como tablas SQL. Nuestro ejemplo sería una consulta ideal para una base de datos, ya que se procesaría como una unidad y requeriría solo una ida y vuelta al servidor de la base de datos.

Sin embargo, esta consulta es ineficiente para una colección local porque la subconsulta se recalcula en cada iteración del bucle externo. Podemos evitar esta ineficiencia ejecutando la subconsulta por separado (de modo que ya no sea una subconsulta):

```
int shortest = names.Min (n => n.Length);

IEnumerable<string> query = from   n in names
                            where  n.Length == shortest
                            select n;
```

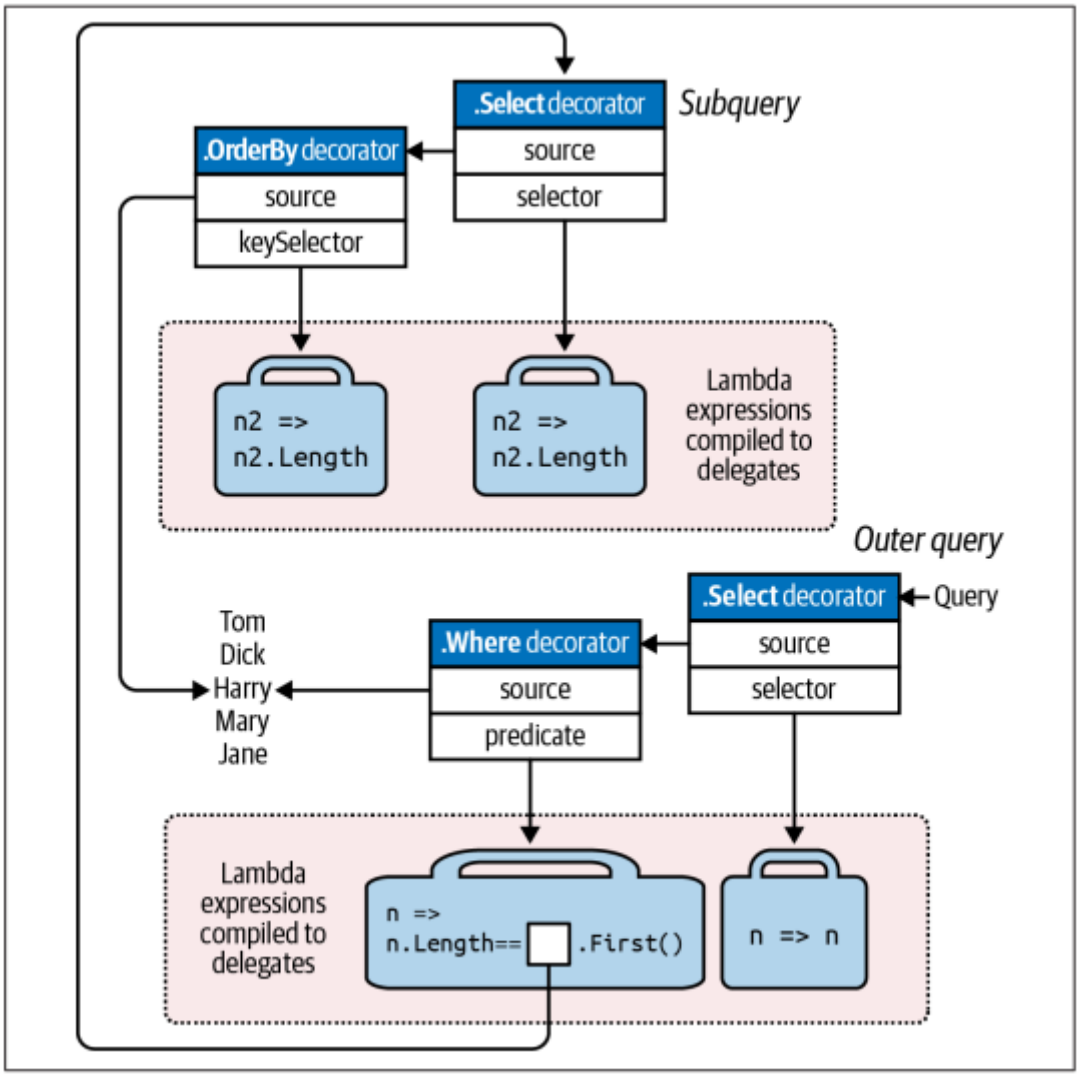


Figure 8-8. UML subquery composition



Factorizar las subconsultas de esta manera casi siempre es deseable al consultar colecciones locales. Una excepción es cuando la subconsulta está correlacionada, lo que significa que hace referencia a la variable de rango externa. Exploraremos las subconsultas correlacionadas en "Proyección" en la página 449.

Subconsultas y ejecución diferida (Subqueries and Deferred Execution)

Un operador de elemento o agregación como First o Count en una subconsulta no fuerza la ejecución inmediata de la consulta externa: la ejecución diferida aún se mantiene para la consulta externa. Esto se debe a que las subconsultas se llaman indirectamente, a través de un delegado en el caso de una consulta local, o a través de un árbol de expresiones en el caso de una consulta interpretada.

Un caso interesante surge cuando se incluye una subconsulta dentro de una expresión Select. En el caso de una consulta local, en realidad estás proyectando una secuencia de consultas, cada

una sujeta a ejecución diferida. El efecto es generalmente transparente y sirve para mejorar aún más la eficiencia. Revisaremos las subconsultas Select con más detalle en el Capítulo 9.

Estrategias de Composición

En esta sección, describiremos tres estrategias para construir consultas más complejas:

- Construcción progresiva de consultas
- Uso de la palabra clave "into"
- Envolver consultas

Todas son estrategias de encadenamiento y producen consultas de tiempo de ejecución idénticas.

Construcción progresiva de consultas

Al comienzo del capítulo, demostramos cómo podrías construir una consulta fluida progresivamente:

```
var filtered   = names    .Where   (n => n.Contains ("a"));
var sorted     = filtered .OrderBy (n => n);
var query      = sorted   .Select  (n => n.ToUpper());
```

Dado que cada uno de los operadores de consulta participantes devuelve una secuencia decoradora, la consulta resultante es la misma cadena o capas de decoradores que obtendrías de una consulta de expresión única. Sin embargo, hay un par de beneficios potenciales al construir consultas progresivamente:

- Puede facilitar la escritura de consultas.
- Puedes agregar operadores de consulta condicionalmente. Por ejemplo,

```
if (includeFilter) query = query.Where (...)
```

es más eficiente que

```
query = query.Where (n => !includeFilter || <expression>)
```

porque evita agregar un operador de consulta adicional si includeFilter es falso.

Un enfoque progresivo a menudo es útil en comprensiones de consultas. Imagina que queremos eliminar todas las vocales de una lista de nombres y luego presentar en orden alfabético aquellos cuya longitud siga siendo mayor a dos caracteres. En sintaxis fluida, podríamos escribir esta consulta como una expresión única, proyectando antes de filtrar:

```
IEnumerable<string> query = names
  .Select  (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", ""))
  .Where   (n => n.Length > 2)
  .OrderBy (n => n);

// Dck
// Hrry
// Mry
```



En lugar de llamar al método Replace de la clase string cinco veces, podríamos eliminar las vocales de una cadena de manera más eficiente utilizando una expresión regular:

```
n => Regex.Replace (n, "[aeiou]", "")
```

Sin embargo, el método Replace de la clase string tiene la ventaja de funcionar también en consultas a bases de datos.

Traducir esto directamente a una expresión de consulta es problemático porque la cláusula select debe venir después de las cláusulas where y orderby. Y si reorganizamos la consulta para proyectar al final, el resultado sería diferente:

```
IEnumerable<string> query =
  from    n in names
  where   n.Length > 2
  orderby n
  select  n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
           .Replace ("o", "").Replace ("u", "");

// Dck
// Hrry
// Jy
// Mry
// Tm
```

Afortunadamente, hay varias formas de obtener el resultado original en la sintaxis de consulta. La primera es mediante la construcción progresiva de la consulta:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "");

query = from n in query where n.Length > 2 orderby n select n;

// Dck
// Hrry
// Mry
```

La palabra clave "into"



La palabra clave "into" se interpreta de dos formas muy diferentes en las expresiones de consulta, dependiendo del contexto. El significado que estamos describiendo ahora es para indicar la continuación de la consulta (el otro significado es para indicar un GroupJoin).

La palabra clave "into" te permite "continuar" una consulta después de una proyección y es un atajo para realizar consultas progresivas. Con "into", podemos reescribir la consulta anterior de la siguiente manera:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "")
  into noVowel
    where noVowel.Length > 2 orderby noVowel select noVowel;
```

El único lugar donde puedes usar "into" es después de una cláusula select o group. "Into" "reinicia" una consulta, permitiéndote introducir nuevas cláusulas where, orderby y select.



Aunque es más fácil pensar en "into" como reiniciar una consulta desde la perspectiva de una expresión de consulta, en su forma final en estilo fluido, todo es una sola consulta. Por lo tanto, no hay un impacto de rendimiento intrínseco con "into". ¡Tampoco pierdes puntos por su uso!

El equivalente de "into" en la sintaxis fluida es simplemente una cadena más larga de operadores.

Reglas de ámbito

Todas las variables de rango están fuera de ámbito después de una palabra clave "into". Lo siguiente no se compilará:

```
var query =
  from n1 in names
  select n1.ToUpper()
  into n2                                  // Only n2 is visible from here on.
    where n1.Contains ("x")                // Illegal: n1 is not in scope.
    select n2;
```

Para entender por qué, considera cómo se mapea esto a la sintaxis fluida:

```
var query = names
  .Select (n1 => n1.ToUpper())
  .Where  (n2 => n1.Contains ("x"));     // Error: n1 no longer in scope
```

El nombre original (n1) se pierde cuando se ejecuta el filtro Where. La secuencia de entrada de Where contiene solo nombres en mayúsculas, por lo que no puede filtrar según n1.

Envolver consultas

Una consulta construida progresivamente se puede formular en una sola declaración al envolver una consulta alrededor de otra. En términos generales:

```
var tempQuery = tempQueryExpr
var finalQuery = from ... in tempQuery ...
```

se puede reformular como:

```
var finalQuery = from ... in (tempQueryExpr)
```

El envoltorio es semánticamente idéntico a la construcción progresiva de consultas o al uso de la palabra clave "into" (sin la variable intermedia). El resultado final en todos los casos es una cadena lineal de operadores de consulta. Por ejemplo, considera la siguiente consulta:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "");

query = from n in query where n.Length > 2 orderby n select n;
```

Reformulado en forma envuelta, queda de la siguiente manera:

```
IEnumerable<string> query =
  from n1 in
  (
    from   n2 in names
    select n2.Replace ("a", "").Replace ("e", "").Replace ("i", "")
            .Replace ("o", "").Replace ("u", "")
  )
  where n1.Length > 2 orderby n1 select n1;
```

Cuando se convierte a sintaxis fluida, el resultado es la misma cadena lineal de operadores que en los ejemplos anteriores:

```
IEnumerable<string> query = names
  .Select  (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", ""))
  .Where   (n => n.Length > 2)
  .OrderBy (n => n);
```

(El compilador no emite el último .Select (n => n), porque es redundante).

Las consultas envueltas pueden resultar confusas porque se asemejan a las subconsultas que escribimos anteriormente. Ambas tienen el concepto de una consulta interna y externa. Sin embargo, al convertirlas a sintaxis fluida, puedes ver que el envoltorio es simplemente una estrategia para encadenar secuencialmente los operadores. El resultado final no se parece en nada a una subconsulta, que incrusta una consulta interna dentro de la expresión lambda de otra.

Volviendo a una analogía anterior: al envolver, la consulta "interna" equivale a las cintas transportadoras anteriores. En contraste, una subconsulta se encuentra por encima de una cinta transportadora y se activa a pedido a través del trabajador lambda de la cinta transportadora (como se ilustra en la Figura 8-7).

Estrategias de proyección

Inicializadores de objetos

Hasta ahora, todas nuestras cláusulas select han proyectado tipos de elementos escalares. Con los inicializadores de objetos de C#, puedes proyectar en tipos más complejos. Por ejemplo, supongamos que, como primer paso en una consulta, queremos eliminar las vocales de una lista

de nombres mientras mantenemos las versiones originales junto a ellas, para beneficio de consultas posteriores. Podemos escribir la siguiente clase para ayudar:

```
class TempProjectionItem
{
  public string Original;    // Original name
  public string Vowelless;   // Vowel-stripped name
}
```

Luego podemos proyectar en él con inicializadores de objetos:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<TempProjectionItem> temp =
  from n in names
  select new TempProjectionItem
  {
    Original  = n,
    Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
  };
```

El resultado es de tipo IEnumerable<TempProjectionItem>, que posteriormente podemos consultar:

```
IEnumerable<string> query = from   item in temp
                            where  item.Vowelless.Length > 2
                            select item.Original;
// Dick
// Harry
// Mary
```

Tipos anónimos

Los tipos anónimos te permiten estructurar tus resultados intermedios sin escribir clases especiales. Podemos eliminar la clase TempProjectionItem en nuestro ejemplo anterior con tipos anónimos:

```
var intermediate = from n in names

  select new
  {
    Original = n,
    Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
  };

IEnumerable<string> query = from   item in intermediate
                            where  item.Vowelless.Length > 2
                            select item.Original;
```

Esto da el mismo resultado que el ejemplo anterior, pero sin necesidad de escribir una clase única. El compilador se encarga de eso, generando una clase temporal con campos que coinciden con la estructura de nuestra proyección. Esto significa, sin embargo, que la consulta intermedia tiene el siguiente tipo:

```
IEnumerable <random-compiler-generated-name>
```

La única forma en que podemos declarar una variable de este tipo es con la palabra clave var. En este caso, var es más que solo un dispositivo para reducir el desorden; es una necesidad.

Podemos escribir toda la consulta de manera más concisa con la palabra clave into:

```
var query = from n in names
  select new
  {
     Original = n,
     Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", "")
  }
  into temp
  where temp.Vowelless.Length > 2
  select temp.Original;
```

Las expresiones de consulta proporcionan un atajo para escribir este tipo de consulta: la palabra clave let.

La palabra clave(keyword) let

La palabra clave let introduce una nueva variable junto con la variable de rango.

Con let, podemos escribir una consulta que extraiga cadenas cuya longitud, excluyendo las vocales, supere los dos caracteres, de la siguiente manera:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query =
  from n in names
  let vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                   .Replace ("o", "").Replace ("u", "")
  where vowelless.Length > 2
  orderby vowelless
  select n;       // Thanks to let, n is still in scope.
```

El compilador resuelve una cláusula let proyectando en un tipo anónimo temporal que contiene tanto la variable de rango como la variable de expresión nueva. En otras palabras, el compilador traduce esta consulta en el ejemplo anterior.

let logra dos cosas:

- Proyecta nuevos elementos junto con los elementos existentes.
- Permite que una expresión se utilice repetidamente en una consulta sin ser reescrita.

El enfoque let es particularmente ventajoso en este ejemplo porque permite que la cláusula select proyecte tanto el nombre original (n) como su versión sin vocales (vowelless).

Puedes tener cualquier número de declaraciones let antes o después de una cláusula where (ver Figura 8-2). Una declaración let puede hacer referencia a variables introducidas en declaraciones let anteriores (sujeto a los límites impuestos por una cláusula into). let reproyecta todas las variables existentes de forma transparente.

Una expresión let no necesita evaluarse como un tipo escalar: a veces es útil que se evalúe como una subsecuencia, por ejemplo.

Consultas Interpretadas (Interpreted Queries)

LINQ proporciona dos arquitecturas paralelas: consultas locales para colecciones de objetos locales y consultas interpretadas para fuentes de datos remotas. Hasta ahora, hemos examinado la arquitectura de las consultas locales, que operan sobre colecciones que implementan IEnumerable<T>. Las consultas locales se resuelven en operadores de consulta en la clase Enumerable (por defecto), que a su vez se resuelven en cadenas de decoradores. Los delegados que aceptan, ya sea expresados en sintaxis de consulta, sintaxis fluida o delegados tradicionales, son completamente locales para el código de Lenguaje de Intermedio (IL), al igual que cualquier otro método de C#.

En cambio, las consultas interpretadas son descriptivas. Operan sobre secuencias que implementan IQueryable<T> y se resuelven en los operadores de consulta en la clase Queryable, que emiten árboles de expresión que se interpretan en tiempo de ejecución. Estos árboles de expresión se pueden traducir, por ejemplo, en consultas SQL, lo que te permite utilizar LINQ para consultar una base de datos.


Los operadores de consulta en Enumerable pueden funcionar con secuencias IQueryable<T>. La dificultad radica en que las consultas resultantes siempre se ejecutan localmente en el cliente. Por esta razón, se proporciona un segundo conjunto de operadores de consulta en la clase Queryable.

Para escribir consultas interpretadas, necesitas comenzar con una API que exponga secuencias de tipo IQueryable<T>. Un ejemplo es Entity Framework Core (EF Core) de Microsoft, que te permite consultar una variedad de bases de datos, incluyendo SQL Server, Oracle, MySQL, PostgreSQL y SQLite.

También es posible generar un envoltorio IQueryable<T> alrededor de una colección enumerable ordinaria llamando al método AsQueryable. Describiremos AsQueryable con más detalle en "Construcción de expresiones de consulta" en la página 442.


IQueryable<T> es una extensión de IEnumerable<T> con métodos adicionales para construir árboles de expresión. La mayoría de las veces puedes ignorar los detalles de estos métodos; son llamados indirectamente por tiempo de ejecución. "Construcción de expresiones de consulta" en la página 442 cubre IQueryable<T> con más detalle.

Para ilustrar, creemos una tabla de clientes simple en SQL Server y la poblamos con algunos nombres utilizando el siguiente script SQL:

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30)
)
insert Customer values (1, 'Tom')
insert Customer values (2, 'Dick')
insert Customer values (3, 'Harry')
insert Customer values (4, 'Mary')
insert Customer values (5, 'Jay')
```

Con esta tabla en su lugar, podemos escribir una consulta LINQ interpretada en C# que utiliza EF Core para recuperar clientes cuyo nombre contiene la letra "a", de la siguiente manera:

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;

using var dbContext = new NutshellContext();

IQueryable<string> query = from c in dbContext.Customers
  where   c.Name.Contains ("a")
  orderby c.Name.Length
  select  c.Name.ToUpper();

foreach (string name in query) Console.WriteLine (name);

public class Customer
{
  public int ID { get; set; }
  public string Name { get; set; }
}

// We'll explain the following class in more detail in the next section.
public class NutshellContext : DbContext
{
  public virtual DbSet<Customer> Customers { get; set; }

  protected override void OnConfiguring (DbContextOptionsBuilder builder)
    => builder.UseSqlServer ("...connection string...");

  protected override void OnModelCreating (ModelBuilder modelBuilder)
    => modelBuilder.Entity<Customer>().ToTable ("Customer")
                                      .HasKey (c => c.ID);
}
```

EF Core traduce esta consulta en el siguiente SQL:

```
SELECT UPPER([c].[Name])
FROM [Customers] AS [c]
WHERE CHARINDEX(N'a', [c].[Name]) > 0
ORDER BY CAST(LEN([c].[Name]) AS int)
```

Aquí está el resultado final:

```
// JAY
// MARY
// HARRY
```

Cómo funcionan las consultas interpretadas

Veamos cómo se procesa la consulta anterior.

Primero, el compilador convierte la sintaxis de consulta en sintaxis fluida. Esto se hace exactamente igual que en las consultas locales:

```
IQueryable<string> query = dbContext.customers
                                    .Where   (n => n.Name.Contains ("a"))
                                    .OrderBy (n => n.Name.Length)
                                    .Select  (n => n.Name.ToUpper());
```

A continuación, el compilador resuelve los métodos de operadores de consulta. Aquí es donde difieren las consultas locales e interpretadas: las consultas interpretadas se resuelven a los operadores de consulta en la clase Queryable en lugar de la clase Enumerable.

Para entender por qué, debemos examinar la variable `dbContext.Customers`, la fuente sobre la cual se construye toda la consulta. `dbContext.Customers` es de tipo `DbSet<T>`, que implementa `IQueryable<T>` (un subtipo de `IEnumerable<T>`). Esto significa que el compilador tiene una opción al resolver `Where`: podría llamar al método de extensión en `Enumerable` o al siguiente método de extensión en `Queryable`:

```
public static IQueryable<TSource> Where<TSource> (this
  IQueryable<TSource> source, Expression <Func<TSource,bool>> predicate)
```

El compilador elige `Queryable.Where` porque su firma es una coincidencia más específica.

`Queryable.Where` acepta un predicado envuelto en un tipo `Expression<TDelegate>`. Esto le indica al compilador que traduzca la expresión lambda proporcionada, es decir, `n=>n.Name.Contains("a")`, a un árbol de expresiones en lugar de un delegado compilado. Un árbol de expresiones es un modelo de objetos basado en los tipos en `System.Linq.Expressions` que se puede inspeccionar en tiempo de ejecución (para que EF Core pueda traducirlo posteriormente a una sentencia SQL).

Dado que `Queryable.Where` también devuelve `IQueryable<T>`, el mismo proceso sigue con los operadores `OrderBy` y `Select`. La Figura 8-9 ilustra el resultado final. En el recuadro sombreado, hay un árbol de expresiones que describe toda la consulta y que se puede recorrer en tiempo de ejecución.

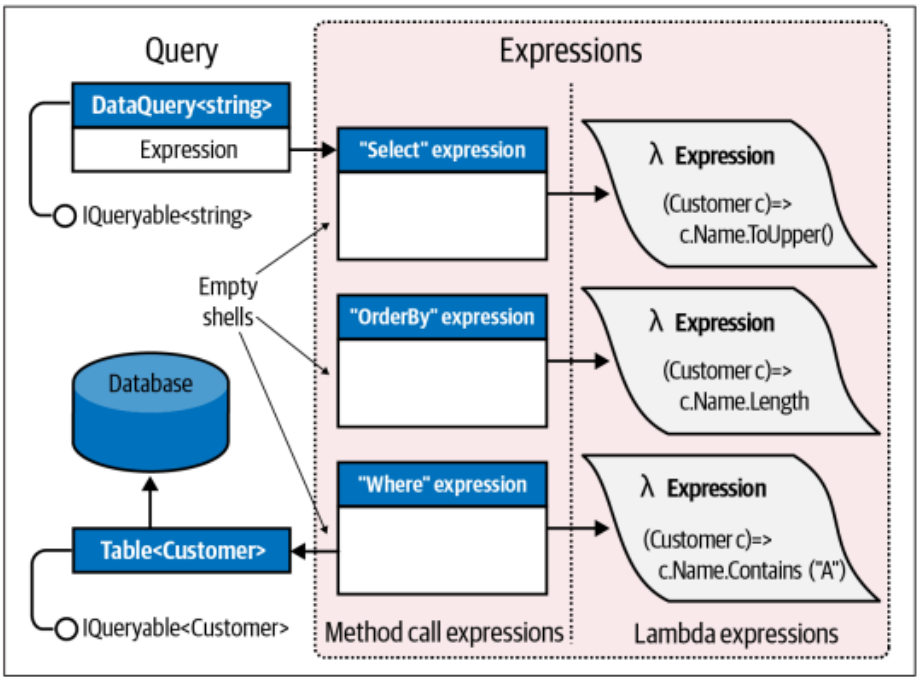


Figure 8-9. Interpreted query composition

Ejecución

Las consultas interpretadas siguen un modelo de ejecución diferida, al igual que las consultas locales. Esto significa que la sentencia SQL no se genera hasta que comienzas a enumerar la consulta. Además, enumerar la misma consulta dos veces resulta en dos consultas al mismo en la base de datos.

Bajo el capó, las consultas interpretadas difieren de las consultas locales en cómo se ejecutan. Cuando enumeras una consulta interpretada, la secuencia más externa ejecuta un programa que recorre todo el árbol de expresiones, procesándolo como una unidad. En nuestro ejemplo, EF Core traduce el árbol de expresiones a una sentencia SQL, que luego ejecuta, devolviendo los resultados como una secuencia.



Para funcionar, EF Core necesita entender el esquema de la base de datos. Esto se logra aprovechando convenciones, atributos de código y una API de configuración fluida. Exploraremos esto en detalle más adelante en el capítulo.

Anteriormente dijimos que una consulta LINQ es como una línea de producción. Sin embargo, cuando enumeras una cinta transportadora IQueryable, no pones en marcha toda la línea de producción, como ocurre con una consulta local. En su lugar, solo se pone en marcha la cinta transportadora IQueryable, con un enumerador especial que llama a un director de producción. El manager revisa toda la línea de producción, que no consiste en código compilado, sino en maniquíes (expresiones de llamada a métodos) con instrucciones pegadas en sus frentes (árboles de expresiones). El manager luego recorre todas las expresiones, en este caso transcribiéndolas en un solo trozo de papel (una sentencia SQL), que luego ejecuta, enviando los

resultados de vuelta al consumidor. Solo una cinta transportadora gira; el resto de la línea de producción es una red de cascarones vacíos, que existen solo para describir lo que se debe hacer.

Esto tiene algunas implicaciones prácticas. Por ejemplo, con consultas locales, puedes escribir tus propios métodos de consulta (bastante fácilmente, con iteradores) y luego usarlos para complementar el conjunto predefinido. Con consultas remotas, esto es difícil e incluso indeseable. Si escribieras un método de extensión MyWhere que aceptara IQueryable<T>, sería como poner tu propio maniquí en la línea de producción. El director de producción no sabría qué hacer con tu maniquí. Incluso si intervinieras en esta etapa, tu solución estaría cableada a un proveedor específico, como EF Core, y no funcionaría con otras implementaciones de IQueryable. Parte del beneficio de tener un conjunto estándar de métodos en Queryable es que definen un vocabulario estándar para consultar cualquier colección remota. Tan pronto como intentes ampliar el vocabulario, ya no serás interoperable.

Otra consecuencia de este modelo es que un proveedor de IQueryable puede ser incapaz de manejar algunas consultas, incluso si te adhieres a los métodos estándar. EF Core está limitado por las capacidades del servidor de la base de datos; algunas consultas de LINQ no tienen traducción a SQL. Si estás familiarizado con SQL, tendrás una buena intuición de cuáles son, aunque a veces tendrás que experimentar para ver qué causa un error en tiempo de ejecución; ¡puede ser sorprendente lo que funciona!

Combinando consultas interpretadas y locales(Combining Interpreted and Local Queries)

Una consulta puede incluir tanto operadores interpretados como locales. Un patrón típico es tener los operadores locales en el exterior y los componentes interpretados en el interior; en otras palabras, las consultas interpretadas alimentan a las consultas locales. Este patrón funciona bien al consultar una base de datos.

Por ejemplo, supongamos que escribimos un método de extensión personalizado para combinar cadenas en una colección:

```
public static IEnumerable<string> Pair (this IEnumerable<string> source)
{
  string firstHalf = null;
  foreach (string element in source)
    if (firstHalf == null)
      firstHalf = element;
    else
    {
      yield return firstHalf + ", " + element;
      firstHalf = null;
    }
}
```

Podemos usar este método de extensión en una consulta que mezcla operadores de EF Core y locales:

```
using var dbContext = new NutshellContext ();
IEnumerable<string> q = dbContext.Customers
  .Select (c => c.Name.ToUpper())
```

```
  .OrderBy (n => n)
  .Pair()                          // Local from this point on.
  .Select ((n, i) => "Pair " + i.ToString() + " = " + n);

foreach (string element in q) Console.WriteLine (element);

// Pair 0 = DICK, HARRY
// Pair 1 = JAY, MARY
```

Debido a que `dbContext.Customers` es de un tipo que implementa `IQueryable<T>`, el operador `Select` se resuelve como `Queryable.Select`. Esto devuelve una secuencia de salida también de tipo `IQueryable<T>`, por lo que el operador `OrderBy` se resuelve de manera similar como `Queryable.OrderBy`. Sin embargo, el siguiente operador de consulta, `Pair`, no tiene una sobrecarga que acepte `IQueryable<T>`, solo la interfaz menos específica `IEnumerable<T>`. Por lo tanto, se resuelve a nuestro método local `Pair`, envolviendo la consulta interpretada en una consulta local. `Pair` también devuelve `IEnumerable`, por lo que el `Select` que le sigue se resuelve a otro operador local.

En el lado de EF Core, la declaración SQL resultante es equivalente a esto:

```
SELECT UPPER([c].[Name]) FROM [Customers] AS [c] ORDER BY UPPER([c].[Name])
```

El resto del trabajo se realiza localmente. En efecto, terminamos con una consulta local (en el exterior) cuya fuente es una consulta interpretada (el interior).

AsEnumerable

`Enumerable.AsEnumerable` es el operador de consulta más simple. Aquí está su definición completa:

```
public static IEnumerable<TSource> AsEnumerable<TSource>
              (this IEnumerable<TSource> source)
{
    return source;
}
```

Su propósito es convertir una secuencia de tipo `IQueryable<T>` a `IEnumerable<T>`, forzando que los operadores de consulta subsiguientes se enlacen con los operadores de `Enumerable` en lugar de los operadores de `Queryable`. Esto hace que el resto de la consulta se ejecute localmente.

Para ilustrar, supongamos que tenemos una tabla `MedicalArticles` en SQL Server y queremos usar EF Core para recuperar todos los artículos sobre influenza cuyo resumen contenga menos de 100 palabras. Para el predicado de palabras, necesitamos una expresión regular:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

using var dbContext = new NutshellContext ();

var query = dbContext.MedicalArticles
  .Where (article => article.Topic == "influenza" &&
                     wordCounter.Matches (article.Abstract).Count < 100);
```

El problema es que SQL Server no admite expresiones regulares, por lo que EF Core arrojará una excepción quejándose de que la consulta no se puede traducir a SQL. Podemos resolver esto realizando la consulta en dos pasos: primero recuperando todos los artículos sobre influenza a través de una consulta de EF Core, y luego filtrando localmente los resúmenes con menos de 100 palabras:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

using var dbContext = new NutshellContext ();

IEnumerable<MedicalArticle> efQuery = dbContext.MedicalArticles
  .Where (article => article.Topic == "influenza");

IEnumerable<MedicalArticle> localQuery = efQuery
  .Where (article => wordCounter.Matches (article.Abstract).Count < 100);
```

Dado que efQuery es de tipo IEnumerable<MedicalArticle>, la segunda consulta se enlaza con los operadores de consulta locales, lo que hace que esa parte del filtrado se ejecute en el cliente.

Con AsEnumerable, podemos hacer lo mismo en una sola consulta:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

using var dbContext = new NutshellContext ();

var query = dbContext.MedicalArticles
  .Where (article => article.Topic == "influenza")

  .AsEnumerable()
  .Where (article => wordCounter.Matches (article.Abstract).Count < 100);
```

Una alternativa a llamar a AsEnumerable es llamar a ToArray o ToList. La ventaja de AsEnumerable es que no fuerza la ejecución inmediata de la consulta ni crea ninguna estructura de almacenamiento.



Mover el procesamiento de la consulta desde el servidor de la base de datos al cliente puede afectar el rendimiento, especialmente si implica recuperar más filas. Una forma más eficiente (aunque más compleja) de resolver nuestro ejemplo sería utilizar la integración SQL CLR para exponer una función en la base de datos que implemente la expresión regular.

Demostraremos aún más consultas combinadas interpretadas y locales en el Capítulo 10.

EF Core

A lo largo de este capítulo y el Capítulo 9, utilizamos EF Core para demostrar las consultas interpretadas. Ahora examinemos las características clave de esta tecnología.

Clases de entidades de EF Core

EF Core te permite usar cualquier clase para representar datos, siempre y cuando contenga una propiedad pública para cada columna que desees consultar.

Por ejemplo, podríamos definir la siguiente clase de entidad para consultar y actualizar una tabla de clientes en la base de datos:

```
public class Customer
{
  public int ID { get; set; }
  public string Name { get; set; }
}
```

DbContext

Después de definir las clases de entidad, el siguiente paso es crear una subclase de DbContext. Una instancia de esta clase representa tus sesiones de trabajo con la base de datos. Por lo general, tu subclase de DbContext contendrá una propiedad DbSet<T> para cada entidad en tu modelo:

```
public class NutshellContext : DbContext
{
  public DbSet<Customer> Customers { get; set; }
  ... properties for other tables ...

}
```

Un objeto DbContext realiza tres funciones:

- Actúa como una fábrica para generar objetos DbSet<> que se pueden consultar.
- Realiza un seguimiento de los cambios que realices en tus entidades para que puedas guardarlos (consultar "Seguimiento de cambios" en la página 437).
- Proporciona métodos virtuales que puedes sobrescribir para configurar la conexión y el modelo.

Configurar la conexión

Al sobrescribir el método OnConfiguring, puedes especificar el proveedor de la base de datos y la cadena de conexión:

```
public class NutshellContext : DbContext
{
  ...
  protected override void OnConfiguring (DbContextOptionsBuilder
                                         optionsBuilder) =>
    optionsBuilder.UseSqlServer
      (@"Server=(local);Database=Nutshell;Trusted_Connection=True");
}
```

En este ejemplo, la cadena de conexión se especifica como una cadena literal. En aplicaciones de producción, normalmente se recupera desde un archivo de configuración como appsettings.json.

UseSqlServer es un método de extensión definido en un ensamblado que forma parte del paquete NuGet Microsoft.EntityFramework.SqlServer. También existen paquetes disponibles para otros proveedores de bases de datos, como Oracle, MySQL, PostgreSQL y SQLite.



Si estás utilizando ASP.NET, puedes permitir que su framework de inyección de dependencias configure previamente optionsBuilder; en la mayoría de los casos, esto te permite evitar la sobreescritura de OnConfiguring por completo. Para habilitar esto, define un constructor en DbContext de la siguiente manera:

```
public NutshellContext (DbContextOptions<NutshellContext>
                        options)
  : base(options) { }
```

Si decides sobrescribir OnConfiguring (quizás para proporcionar una configuración si tu DbContext se utiliza en otro escenario), puedes verificar si las opciones ya se han configurado de la siguiente manera:

```
protected override void OnConfiguring (
  DbContextOptionsBuilder optionsBuilder)
{
  if (!optionsBuilder.IsConfigured)
  {
    ...
  }
}
```

En el método OnConfiguring, también puedes habilitar otras opciones, incluida la carga perezosa (ver "Carga perezosa" en la página 440).

Configurando el Modelo

Por defecto, EF Core utiliza convenciones para inferir el esquema de la base de datos a partir de los nombres de tus clases y propiedades.

Puedes anular las configuraciones predeterminadas utilizando el fluente API al sobrescribir OnModelCreating y llamar a los métodos de extensión en el parámetro ModelBuilder. Por ejemplo, podemos especificar explícitamente el nombre de la tabla de base de datos para nuestra entidad Customer de la siguiente manera:

```
protected override void OnModelCreating (ModelBuilder modelBuilder) =>
  modelBuilder.Entity<Customer>()
    .ToTable ("Customer");   // Table is called 'Customer'
```

Sin este código, EF Core mapearía esta entidad a una tabla llamada "Customers" en lugar de "Customer", debido a que tenemos una propiedad DbSet<Customer> en nuestro DbContext llamada Customers:

```
public DbSet<Customer> Customers { get; set; }
```



El siguiente código asigna todas las entidades a nombres de tabla que coinciden con el nombre de la clase de entidad (que normalmente está en singular) en lugar del nombre de la propiedad DbSet<T> (que normalmente está en plural):

```
protected override void OnModelCreating (ModelBuilder
                                         modelBuilder)
{
  foreach (IMutableEntityType entityType in
           modelBuilder.Model.GetEntityTypes())
  {
    modelBuilder.Entity (entityType.Name)
      .ToTable (entityType.ClrType.Name);
  }
}
```

El fluent API ofrece una sintaxis ampliada para configurar columnas. En el siguiente ejemplo, utilizamos dos métodos populares:

- HasColumnName, que asigna una propiedad a una columna con un nombre diferente.
- IsRequired, que indica que una columna no puede ser nula.

```
protected override void OnModelCreating (ModelBuilder modelBuilder) =>
  modelBuilder.Entity<Customer> (entity =>
  {
      entity.ToTable ("Customer");
      entity.Property (e => e.Name)
            .HasColumnName ("Full Name")  // Column name is 'Full Name'
            .IsRequired();                // Column is not nullable
  });
```

La Tabla 8-1 muestra algunos de los métodos más importantes del fluent API.



En lugar de utilizar el fluent API, también puedes configurar tu modelo aplicando atributos especiales a las clases de entidad y propiedades ("anotaciones de datos"). Sin embargo, esta aproximación es menos flexible, ya que la configuración debe realizarse en tiempo de compilación, y menos potente, ya que hay algunas opciones que solo se pueden configurar mediante el fluent API.

Table 8-1. Fluent API model configuration methods

Method	Purpose	Example
`ToTable`	Specify the database table name for a given entity	`builder .Entity<Customer>() .ToTable("Customer");`
`HasColumnName`	Specify the column name for a given property	`builder.Entity<Customer>() .Property(c => c.Name) .HasColumnName("Full Name");`
`HasKey`	Specify a key (usually that deviates from convention)	`builder.Entity<Customer>() .HasKey(c => c.CustomerNr);`

Method	Purpose	Example

IsRequired	Specify that the property requires a value (is not nullable)	`builder.Entity<Customer>()`
		`  .Property(c => c.Name)`
		`  .IsRequired();`

HasMaxLength	Specify the maximum length of a variable-length type (usually a string) whose width can vary	`builder.Entity<Customer>()`
		`  .Property(c => c.Name)`
		`  .HasMaxLength(60);`

HasColumnType	Specify the database data type for a column	`builder.Entity<Purchase>()`
		`  .Property(p => p.Description)`
		`  .HasColumnType("varchar(80)");`

Ignore	Ignore a type	`builder.Ignore<Products>();`
Ignore	Ignore a property of a type	`builder.Entity<Customer>()` `  .Ignore(c => c.ChatName);`

HasIndex	Specify a property (or combination of properties) should serve in the database as an index	`// Compound index:`
		`builder.Entity<Purchase>()`
		`  .HasIndex(p =>`
		`  new { p.Date, p.Price });`
		
		`// Unique index on one property`
		`builder`
		`  .Entity<MedicalArticle>()`
		`  .HasIndex(a => a.Topic)`
		`  .IsUnique();`

HasOne	See "Navigation Properties" on page 438	`builder.Entity<Purchase>()`
		`  .HasOne(p => p.Customer)`
		`  .WithMany(c => c.Purchases);`

HasMany	See "Navigation Properties" on page 438	`builder.Entity<Customer>()`
		`  .HasMany(c => c.Purchases)`
		`  .WithOne(p => p.Customer);`

Crear la base de datos (Creating the database)

EF Core admite un enfoque de "code-first", lo que significa que puedes comenzar definiendo las clases de entidad y luego pedirle a EF Core que cree la base de datos. La forma más sencilla de hacerlo es llamando al siguiente método en una instancia de DbContext:

```
dbContext.Database.EnsureCreated();
```

Sin embargo, un enfoque mejor es utilizar la función de migraciones de EF Core, que no solo crea la base de datos, sino que también la configura de manera que EF Core pueda actualizar automáticamente el esquema en el futuro cuando cambien las clases de entidad. Puedes habilitar las migraciones en la Consola del Administrador de paquetes de Visual Studio y pedirle que cree la base de datos con los siguientes comandos:

```
Install-Package Microsoft.EntityFrameworkCore.Tools
Add-Migration InitialCreate
Update-Database
```

El primer comando instala las herramientas para administrar EF Core desde Visual Studio.

El segundo comando genera una clase especial en C# conocida como migración de código que contiene instrucciones para crear la base de datos. El último comando ejecuta esas instrucciones contra la cadena de conexión de la base de datos especificada en el archivo de configuración de la aplicación del proyecto.

Usando DbContext

Después de haber definido las clases de entidad y haber subclasificado DbContext, puedes instanciar tu DbContext y consultar la base de datos de la siguiente manera:

```
using var dbContext = new NutshellContext();
Console.WriteLine (dbContext.Customers.Count());
// Executes "SELECT COUNT(*) FROM [Customer] AS [c]"
```

También puedes usar la instancia de tu DbContext para escribir en la base de datos. El siguiente código inserta una fila en la tabla Customer:

```
using var dbContext = new NutshellContext();
Customer cust = new Customer()
{
  Name = "Sara Wells"
};
dbContext.Customers.Add (cust);
dbContext.SaveChanges();    // Writes changes back to database
```

El siguiente código consulta la base de datos para el cliente que acaba de ser insertado:

```
using var dbContext = new NutshellContext();
Customer cust = dbContext.Customers
  .Single (c => c.Name == "Sara Wells")
```

El siguiente código actualiza el nombre de ese cliente y guarda el cambio en la base de datos:

```
cust.Name = "Dr. Sara Wells";
dbContext.SaveChanges();
```



El operador Single es ideal para recuperar una fila por clave primaria. A diferencia de First, arroja una excepción si se devuelve más de un elemento.

Seguimiento de objetos

Una instancia de DbContext realiza un seguimiento de todas las entidades que instancia, por lo que puede devolverte las mismas entidades cada vez que solicites las mismas filas de una tabla. En otras palabras, un contexto nunca emitirá dos entidades separadas que se refieran a la misma fila en una tabla (donde una fila se identifica por clave primaria). Esta capacidad se llama seguimiento de objetos.

Para ilustrar, supongamos que el cliente cuyo nombre es el primero alfabéticamente también tiene el ID más bajo. En el siguiente ejemplo, "a" y "b" harán referencia al mismo objeto:

```
using var dbContext = new NutshellContext ();

Customer a = dbContext.Customers.OrderBy (c => c.Name).First();
Customer b = dbContext.Customers.OrderBy (c => c.ID).First();
```

Desechando DbContext (Disposing DbContext)

Aunque DbContext implementa IDisposable, en general puedes prescindir de desechar las instancias. El proceso de desecho fuerza el cierre de la conexión del contexto, pero esto generalmente no es necesario porque EF Core cierra las conexiones automáticamente cuando finalizas la recuperación de resultados de una consulta.

Desechar un contexto prematuramente puede ser problemático debido a la evaluación diferida (lazy evaluation). Considera el siguiente ejemplo:

```
IQueryable<Customer> GetCustomers (string prefix)
{
  using (var dbContext = new NutshellContext ())
    return dbContext.Customers
                    .Where (c => c.Name.StartsWith (prefix));
}
...
foreach (Customer c in GetCustomers ("a"))
  Console.WriteLine (c.Name);
```

Esto fallará porque la consulta se evalúa cuando la enumeramos, lo cual ocurre después de que se ha desechado el DbContext.

Sin embargo, hay algunas advertencias sobre no desechar los contextos:

- Se basa en que el objeto de conexión libere todos los recursos no administrados en el método Close. Aunque esto es cierto con SqlConnection, teóricamente es posible que una conexión de un proveedor externo mantenga recursos abiertos si llamas a Close pero no a Dispose (aunque esto violaría potencialmente el contrato definido por IDbConnection.Close).

- Si llamas manualmente a GetEnumerator en una consulta (en lugar de usar foreach) y luego no desechas el enumerador ni consumes la secuencia, la conexión permanecerá abierta. El desecho del DbContext proporciona una salvaguardia en tales escenarios.

- Algunas personas consideran que es más ordenado deshacer los contextos (y todos los objetos que implementan IDisposable).

Si deseas desechar explícitamente los contextos, debes pasar una instancia de DbContext a métodos como GetCustomers para evitar el problema descrito. En escenarios como ASP.NET Core MVC, donde la instancia del contexto se proporciona mediante inyección de dependencias (DI), la infraestructura de DI se encargará del ciclo de vida del contexto. Se creará cuando comience una unidad de trabajo (como una solicitud HTTP procesada en el controlador) y se desechará cuando esa unidad de trabajo finalice.

Consideremos qué sucede cuando EF Core encuentra la segunda consulta. Comienza consultando la base de datos y obtiene una sola fila. Luego lee la clave primaria de esta fila y realiza una búsqueda en la caché de entidades del contexto. Al encontrar una coincidencia,

devuelve el objeto existente sin actualizar ningún valor. Por lo tanto, si otro usuario acaba de actualizar el nombre de ese cliente en la base de datos, se ignorará el nuevo valor. Esto es esencial para evitar efectos secundarios inesperados (el objeto Customer podría estar en uso en otro lugar) y también para gestionar la concurrencia. Si hubieras modificado propiedades en el objeto Customer y aún no hubieras llamado a SaveChanges, no querrías que tus propiedades se sobrescriban automáticamente.



Puedes desactivar el seguimiento de objetos agregando el método de extensión AsNoTracking a tu consulta o configurando la propiedad ChangeTracker.QueryTrackingBehavior en el contexto con el valor QueryTrackingBehavior.NoTracking. Las consultas sin seguimiento son útiles cuando los datos se utilizan solo para lectura, ya que mejora el rendimiento y reduce el uso de memoria.

Para obtener información actualizada de la base de datos, debes instanciar un nuevo contexto o llamar al método Reload de la siguiente manera:

```
dbContext.Entry (myCustomer).Reload();
```

La mejor práctica es utilizar una nueva instancia de DbContext por cada unidad de trabajo para que la necesidad de recargar manualmente una entidad sea poco frecuente.

Seguimiento de cambios

Cuando cambias el valor de una propiedad en una entidad cargada a través de DbContext, EF Core reconoce el cambio y actualiza la base de datos en consecuencia al llamar a SaveChanges. Para hacerlo, crea una instantánea del estado de las entidades cargadas a través de tu subclase de DbContext y compara el estado actual con el estado original cuando se llama a SaveChanges (o cuando consultas manualmente el seguimiento de cambios, como verás a continuación).

Puedes enumerar los cambios rastreados en un DbContext de la siguiente manera:

```
foreach (var e in dbContext.ChangeTracker.Entries())
{
  Console.WriteLine ($"{e.Entity.GetType().FullName} is {e.State}");
  foreach (var m in e.Members)
    Console.WriteLine (
      $"  {m.Metadata.Name}: '{m.CurrentValue}' modified: {m.IsModified}");
}
```

Cuando llamas a SaveChanges, EF Core utiliza la información del ChangeTracker para construir instrucciones SQL que actualizarán la base de datos para que coincida con los cambios en tus objetos, emitiendo instrucciones de inserción para agregar nuevas filas, instrucciones de actualización para modificar datos e instrucciones de eliminación para eliminar filas que se hayan eliminado del grafo de objetos en tu subclase de DbContext. Se respeta cualquier TransactionScope; si no hay ninguna presente, envuelve todas las declaraciones en una nueva transacción.

Puedes optimizar el seguimiento de cambios implementando INotifyPropertyChanged y, opcionalmente, INotifyPropertyChanging en tus entidades. El primero permite a EF Core evitar la sobrecarga de comparar entidades modificadas con las originales; el segundo permite a EF Core evitar almacenar los valores originales por completo. Después de implementar estas interfaces, llama al método HasChangeTrackingStrategy en el ModelBuilder al configurar el modelo para activar el seguimiento de cambios optimizado.

Propiedades de navegación

Las propiedades de navegación te permiten hacer lo siguiente:

- Consultar tablas relacionadas sin tener que unir manualmente.
- Insertar, eliminar y actualizar filas relacionadas sin actualizar explícitamente las claves foráneas.

Por ejemplo, supongamos que un cliente puede tener varias compras. Podemos representar una relación uno a muchos entre Cliente y Compra con las siguientes entidades:

```
public class Customer
{
  public int ID { get; set; }
  public string Name { get; set; }

  // Child navigation property, which must be of type ICollection<T>:
  public virtual List<Purchase> Purchases {get;set;} = new List<Purchase>();
}

public class Purchase
{
  public int ID { get; set; }
  public DateTime Date { get; set; }
  public string Description { get; set; }
  public decimal Price { get; set; }
  public int CustomerID? { get; set; }     // Foreign key field

  public Customer Customer { get; set; }   // Parent navigation property
}
```

EF Core es capaz de inferir a partir de estas entidades que ClienteID es una clave foránea a la tabla Cliente, porque el nombre "ClienteID" sigue una convención de nombres popular. Si le pidiéramos a EF Core que cree una base de datos a partir de estas entidades, crearía una restricción de clave foránea entre Purchase.CustomerID y Customer.ID.



Si EF Core no puede inferir la relación, puedes configurarla explícitamente en el método OnModelCreating de la siguiente manera:

```
modelBuilder.Entity<Purchase>()
  .HasOne (e => e.Customer)
  .WithMany (e => e.Purchases)
  .HasForeignKey (e => e.CustomerID);
```

Con estas propiedades de navegación configuradas, podemos escribir consultas como esta:

```
var customersWithPurchases = Customers.Where (c => c.Purchases.Any());
```

En el Capítulo 9 se cubre en detalle cómo escribir dichas consultas.

Agregar y eliminar entidades de colecciones de navegación

Cuando agregas nuevas entidades a una propiedad de navegación de colección, EF Core automáticamente completa las claves foráneas al llamar a SaveChanges:

```
Customer cust = dbContext.Customers.Single (c => c.ID == 1);

Purchase p1 = new Purchase { Description="Bike",  Price=500 };
Purchase p2 = new Purchase { Description="Tools", Price=100 };

cust.Purchases.Add (p1);
cust.Purchases.Add (p2);

dbContext.SaveChanges();
```

En este ejemplo, EF Core escribe automáticamente 1 en la columna CustomerID de cada una de las nuevas compras y escribe el ID generado por la base de datos para cada compra en Purchase.ID.

Cuando eliminas una entidad de una propiedad de navegación de colección y llamas a SaveChanges, EF Core borrará el campo de clave foránea o eliminará la fila correspondiente de la base de datos, dependiendo de cómo se haya configurado o inferido la relación. En este caso, hemos definido Purchase.CustomerID como un entero nullable (para representar compras sin un cliente o transacciones en efectivo), por lo que eliminar una compra de un cliente borraría el campo de clave foránea en lugar de eliminarlo de la base de datos.

Cargar propiedades de navegación

Cuando EF Core carga una entidad, por defecto no carga sus propiedades de navegación:

```
using var dbContext = new NutshellContext();
var cust = dbContext.Customers.First();
Console.WriteLine (cust.Purchases.Count);    // Always 0
```

Una solución es utilizar el método de extensión Include, que le indica a EF Core que cargue de forma ansiosa las propiedades de navegación:

```
var cust = dbContext.Customers
  .Include (c => c.Purchases)
  .Where (c => c.ID == 2).First();
```

Otra solución es utilizar una proyección. Esta técnica es particularmente útil cuando solo necesitas trabajar con algunas de las propiedades de la entidad, ya que reduce la transferencia de datos:

```
var custInfo = dbContext.Customers
  .Where (c => c.ID == 2)
  .Select (c => new
    {
      Name = c.Name,

    Purchases = c.Purchases.Select (p => new { p.Description, p.Price })
  })
.First();
```

Ambas técnicas informan a EF Core qué datos necesitas para que puedan ser obtenidos en una única consulta a la base de datos. También es posible instruir manualmente a EF Core para que cargue una propiedad de navegación según sea necesario:

```
dbContext.Entry (cust).Collection (b => b.Purchases).Load();
// cust.Purchases is now populated.
```

Esto se conoce como carga explícita. A diferencia de los enfoques anteriores, esto genera una consulta adicional a la base de datos.

Carga perezosa (Lazy loading)

Otro enfoque para cargar propiedades de navegación se llama carga perezosa (lazy loading). Cuando está habilitada, EF Core carga las propiedades de navegación bajo demanda mediante la generación de una clase proxy para cada una de tus clases de entidad, la cual intercepta los intentos de acceso a propiedades de navegación no cargadas. Para que esto funcione, cada propiedad de navegación debe ser virtual y la clase en la que está definida debe ser heredable (no sellada). Además, el contexto no debe haber sido eliminado (disposed) cuando ocurre la carga perezosa, para que se pueda realizar una solicitud adicional a la base de datos.

Puedes habilitar la carga perezosa en el método OnConfiguring de tu clase DbContext de la siguiente manera:

```
protected override void OnConfiguring (DbContextOptionsBuilder
                                       optionsBuilder)
{
  optionsBuilder
    .UseLazyLoadingProxies()
    ...
}
```

(También deberás agregar una referencia al paquete NuGet Microsoft.EntityFrameworkCore.Proxies).

El costo de la carga perezosa es que EF Core debe realizar una solicitud adicional a la base de datos cada vez que accedes a una propiedad de navegación no cargada. Si realizas muchas de estas solicitudes, el rendimiento puede verse afectado debido a un exceso de ida y vuelta con la base de datos.



Con la carga perezosa habilitada, el tipo en tiempo de ejecución de tus clases es un proxy derivado de tu clase de entidad. Por ejemplo:

```
using var dbContext = new NutshellContext();
var cust = dbContext.Customers.First();
Console.WriteLine (cust.GetType());
// Castle.Proxies.CustomerProxy
```

Ejecución aplazada (Deferred Execution)

Las consultas de EF Core están sujetas a ejecución aplazada, al igual que las consultas locales. Esto te permite construir consultas de forma progresiva. Sin embargo, hay un aspecto en el que EF Core tiene una semántica especial de ejecución diferida cuando aparece una subconsulta dentro de una expresión Select.

Con consultas locales, se obtiene una doble ejecución diferida, ya que desde una perspectiva funcional, estás seleccionando una secuencia de consultas. Entonces, si enumeras la secuencia de resultados externos pero nunca enumeras las secuencias internas, la subconsulta nunca se ejecutará.

Con EF Core, la subconsulta se ejecuta al mismo tiempo que la consulta principal externa. Esto evita excesivos viajes de ida y vuelta.

Por ejemplo, la siguiente consulta se ejecuta en un solo viaje de ida y vuelta al alcanzar la primera declaración foreach:

```
using var dbContext = new NutshellContext ();

var query = from c in dbContext.Customers
            select
               from p in c.Purchases
               select new { c.Name, p.Price };

foreach (var customerPurchaseResults in query)
  foreach (var namePrice in customerPurchaseResults)
    Console.WriteLine ($"{ namePrice.Name} spent { namePrice.Price}");
```

Cualquier propiedad de navegación que proyectes explícitamente se carga completamente en un solo viaje de ida y vuelta:

```
var query = from c in dbContext.Customers
            select new { c.Name, c.Purchases };

foreach (var row in query)
  foreach (Purchase p in row.Purchases)   // No extra round-tripping
    Console.WriteLine (row.Name + " spent " + p.Price);
```

Pero si enumeramos una propiedad de navegación sin haberla cargado previamente o proyectado, se aplican las reglas de ejecución diferida. En el siguiente ejemplo, EF Core ejecuta otra consulta de "Purchases" en cada iteración del bucle (suponiendo que la carga diferida está habilitada):

```
foreach (Customer c in dbContext.Customers.ToArray())
  foreach (Purchase p in c.Purchases)    // Another SQL round-trip
    Console.WriteLine (c.Name + " spent " + p.Price);
```

Este modelo es ventajoso cuando deseas ejecutar selectivamente el bucle interno, basado en una prueba que solo se puede realizar en el cliente:

```
foreach (Customer c in dbContext.Customers.ToArray())
  if (myWebService.HasBadCreditHistory (c.ID))
    foreach (Purchase p in c.Purchases)   // Another SQL round trip
      Console.WriteLine (c.Name + " spent " + p.Price);
```



Toma nota del uso de `ToArray` en las consultas anteriores. Por defecto, SQL Server no puede iniciar una nueva consulta mientras aún se están procesando los resultados de la consulta actual. Llamar a `ToArray` materializa los clientes para que se puedan emitir consultas adicionales para recuperar las compras por cliente. Es posible configurar SQL Server para permitir múltiples conjuntos de resultados activos (MARS) al agregar `;MultipleActiveResultSets=True` a la cadena de conexión de la base de datos. Utiliza MARS con precaución, ya que puede ocultar un diseño de base de datos que genera muchas consultas y que podría mejorarse cargando de forma anticipada y/o proyectando los datos requeridos.

(En el Capítulo 9, exploraremos las subconsultas en más detalle en la sección "Proyectando" en la página 449.)

Construyendo Expresiones de Consulta

Hasta ahora, en este capítulo, cuando hemos necesitado componer consultas dinámicamente, lo hemos hecho encadenando operadores de consulta de forma condicional. Aunque esto es adecuado en muchos escenarios, a veces necesitas trabajar a un nivel más granular y componer dinámicamente las expresiones lambda que alimentan a los operadores.

En esta sección, asumimos la siguiente clase `Product`:

```
public class Product
{
  public int ID { get; set; }
  public string Description { get; set; }
  public bool Discontinued { get; set; }
  public DateTime LastSale { get; set; }
}
```

Delegados versus Árboles de Expresión

Recuerda que:

- Las consultas locales, que utilizan operadores `Enumerable`, utilizan delegados.
- Las consultas interpretadas, que utilizan operadores `Queryable`, utilizan árboles de expresión.

Podemos ver esto al comparar la firma del operador `Where` en `Enumerable` y `Queryable`:

```
public static IEnumerable<TSource> Where<TSource> (this
  IEnumerable<TSource> source, Func<TSource,bool> predicate)

public static IQueryable<TSource> Where<TSource> (this
  IQueryable<TSource> source, Expression<Func<TSource,bool>> predicate)
```

Cuando se incorpora en una consulta, una expresión lambda se ve idéntica, ya sea que se vincule a los operadores de `Enumerable` o a los operadores de `Queryable`:

```
IEnumerable<Product> q1 = localProducts.Where (p => !p.Discontinued);
IQueryable<Product>  q2 = sqlProducts.Where   (p => !p.Discontinued);
```

Cuando asignas una expresión lambda a una variable intermedia, sin embargo, debes ser explícito acerca de si quieres que se resuelva como un delegado (es decir, `Func<>`) o como un árbol de expresión (es decir, `Expression<Func<>>`). En el siguiente ejemplo, `predicate1` y `predicate2` no son intercambiables:

```
Func <Product, bool> predicate1 = p => !p.Discontinued;
IEnumerable<Product> q1 = localProducts.Where (predicate1);

Expression <Func <Product, bool>> predicate2 = p => !p.Discontinued;
IQueryable<Product> q2 = sqlProducts.Where (predicate2);
```

Compilación de árboles de expresión

Puedes convertir un árbol de expresión a un delegado llamando a `Compile`. Esto es particularmente útil al escribir métodos que devuelven expresiones reutilizables. Para ilustrar esto, agreguemos un método estático a la clase `Product` que devuelve un predicado que evalúa a verdadero si un producto no ha sido descontinuado y se ha vendido en los últimos 30 días:

```
public class Product
{
  public static Expression<Func<Product, bool>> IsSelling()
  {
    return p => !p.Discontinued && p.LastSale > DateTime.Now.AddDays (-30);
  }
}
```

El método que acabamos de escribir se puede usar tanto en consultas interpretadas como en consultas locales de la siguiente manera:

```
void Test()
{
  var dbContext = new NutshellContext();
  Product[] localProducts = dbContext.Products.ToArray();

  IQueryable<Product> sqlQuery =
    dbContext.Products.Where (Product.IsSelling());

  IEnumerable<Product> localQuery =
    localProducts.Where (Product.IsSelling().Compile());
}
```



.NET no proporciona una API para convertir en la dirección inversa, de un delegado a un árbol de expresión. Esto hace que los árboles de expresión sean más versátiles.

Operador AsQueryable

El operador AsQueryable te permite escribir consultas completas que pueden ejecutarse sobre secuencias locales o remotas:

```
IQueryable<Product> FilterSortProducts (IQueryable<Product> input)
{
  return from p in input
         where ...

          orderby ...
          select p;
 }

 void Test()
 {
   var dbContext = new NutshellContext();
   Product[] localProducts = dbContext.Products.ToArray();

   var sqlQuery   = FilterSortProducts (dbContext.Products);
   var localQuery = FilterSortProducts (localProducts.AsQueryable());
   ...
 }
```

El operador AsQueryable envuelve una secuencia local con IQueryable<T> para que los operadores de consulta subsiguientes se resuelvan en árboles de expresión. Cuando posteriormente enumeras el resultado, los árboles de expresión se compilan implícitamente (a un pequeño costo de rendimiento) y la secuencia local se enumera como lo haría normalmente.

Árboles de Expresión

Dijimos anteriormente que una conversión implícita de una expresión lambda a Expression<TDelegate> hace que el compilador de C# emita código que construye un árbol de expresión. Con un poco de esfuerzo de programación, puedes hacer lo mismo manualmente en tiempo de ejecución, es decir, construir dinámicamente un árbol de expresión desde cero. El resultado se puede convertir a Expression<TDelegate> y usar en consultas de EF Core o compilarlo en un delegado ordinario llamando a Compile.

El DOM de Expresiones

Un árbol de expresión es un mini-DOM de código. Cada nodo en el árbol está representado por un tipo en el espacio de nombres System.Linq.Expressions. La Figura 8-10 ilustra estos tipos.

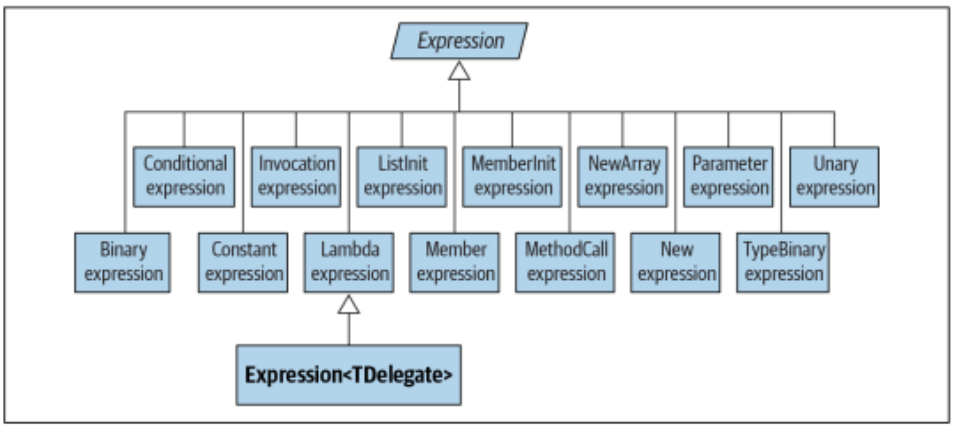


Figure 8-10. Expression types

La clase base para todos los nodos es la clase Expression (no genérica). La clase genérica Expression<TDelegate> en realidad significa "expresión lambda tipada" y podría haberse llamado LambdaExpression<TDelegate> si no fuera por la torpeza de esto:

```
LambdaExpression<Func<Customer,bool>> f = ...
```

La clase base de Expression<T> es la clase LambdaExpression (no genérica). LambdaExpression proporciona unificación de tipos para los árboles de expresión lambda: cualquier Expression<T> tipado se puede convertir a LambdaExpression.

Lo que distingue a las LambdaExpressions de las Expressions ordinarias es que las expresiones lambda tienen parámetros.

Para crear un árbol de expresión, no instancies directamente los tipos de nodo; en su lugar, llama a los métodos estáticos proporcionados en la clase Expression, como Add, And, Call, Constant, LessThan, y así sucesivamente.

La Figura 8-11 muestra el árbol de expresión que crea la siguiente asignación:

```
Expression<Func<string, bool>> f = s => s.Length < 5;
```

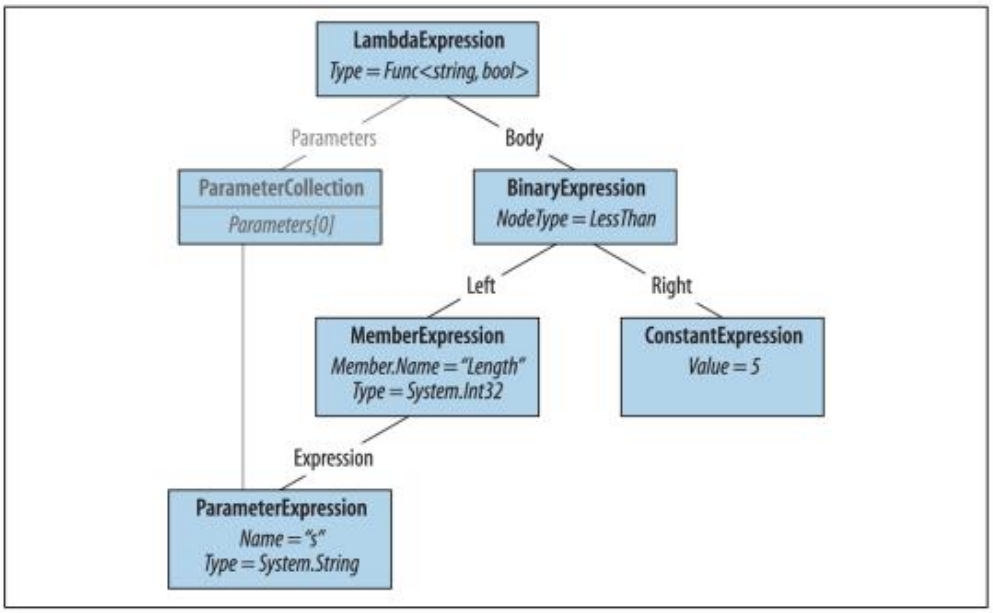


Figure 8-11. Expression tree

Podemos demostrar esto de la siguiente manera:

```
Console.WriteLine (f.Body.NodeType);                    // LessThan
Console.WriteLine (((BinaryExpression) f.Body).Right);  // 5
```

Ahora construyamos esta expresión desde cero. El principio es que empiezas desde la parte inferior del árbol y avanzas hacia arriba. Lo más bajo en nuestro árbol es un ParameterExpression, el parámetro de la expresión lambda llamado "s" de tipo string:

```
ParameterExpression p = Expression.Parameter (typeof (string), "s");
```

El siguiente paso es construir el MemberExpression y el ConstantExpression. En el primer caso, necesitamos acceder a la propiedad Length de nuestro parámetro "s":

```
MemberExpression stringLength = Expression.Property (p, "Length");
ConstantExpression five = Expression.Constant (5);
```

Luego viene la comparación LessThan:

```
BinaryExpression comparison = Expression.LessThan (stringLength, five);
```

El último paso es construir la expresión lambda, que vincula un cuerpo de expresión a una colección de parámetros:

```
Expression<Func<string, bool>> lambda
  = Expression.Lambda<Func<string, bool>> (comparison, p);
```

Una forma conveniente de probar nuestra expresión lambda es compilarla en un delegado:

```
Func<string, bool> runnable = lambda.Compile();

Console.WriteLine (runnable ("kangaroo"));          // False
Console.WriteLine (runnable ("dog"));               // True
```



La forma más sencilla de determinar qué tipo de expresión utilizar es examinar una expresión lambda existente en el depurador de Visual Studio.

Continuaremos esta discusión en línea en http://www.albahari.com/expressions.

9 Operadores de LINQ

Este capítulo describe cada uno de los operadores de consulta de LINQ. Además de servir como referencia, dos de las secciones, "Proyección" en la página 449 y "Unión" en la página 449, cubren varias áreas conceptuales:

- Proyección de jerarquías de objetos
- Unión con Select, SelectMany, Join y GroupJoin
- Expresiones de consulta con múltiples variables de rango

Todos los ejemplos en este capítulo asumen que se define una matriz de nombres de la siguiente manera:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

Los ejemplos que consultan una base de datos asumen que se instancia una variable llamada dbContext de la siguiente manera:

```
var dbContext = new NutshellContext();
```

donde NutshellContext está definido de la siguiente manera:

```
public class NutshellContext : DbContext
{
  public DbSet<Customer> Customers { get; set; }
  public DbSet<Purchase> Purchases { get; set; }

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Customer>(entity =>
    {
      entity.ToTable("Customer");
      entity.Property(e => e.Name).IsRequired();  // Column is not nullable
    });
    modelBuilder.Entity<Purchase>(entity =>
    {
      entity.ToTable("Purchase");
      entity.Property(e => e.Date).IsRequired();
```

```
        entity.Property(e => e.Description).IsRequired();
      });
    }
  }

  public class Customer
  {
    public int ID { get; set; }
    public string Name { get; set; }

    public virtual List<Purchase> Purchases { get; set; }
      = new List<Purchase>();
  }

  public class Purchase
  {
    public int ID { get; set; }
    public int? CustomerID { get; set; }
    public DateTime Date { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }

    public virtual Customer Customer { get; set; }
  }
```



Todos los ejemplos de este capítulo están precargados en LINQPad, junto con una base de datos de muestra con un esquema correspondiente. Puedes descargar LINQPad desde http://www.linqpad.net.

Aquí están las definiciones de tabla correspondientes para SQL Server:

```
CREATE TABLE Customer (
  ID int NOT NULL IDENTITY PRIMARY KEY,
  Name nvarchar(30) NOT NULL
)

CREATE TABLE Purchase (
  ID int NOT NULL IDENTITY PRIMARY KEY,
  CustomerID int NOT NULL REFERENCES Customer(ID),
  Date datetime NOT NULL,
  Description nvarchar(30) NOT NULL,
  Price decimal NOT NULL
)
```

Resumen

En esta sección, proporcionamos una visión general de los operadores de consulta estándar. Se dividen en tres categorías:

- Secuencia de entrada, secuencia de salida (secuencia → secuencia)
- Secuencia de entrada, elemento único o valor escalar de salida
- Sin entrada, secuencia de salida (métodos de generación)

Primero presentamos cada una de las tres categorías y los operadores de consulta que incluyen, y luego abordamos cada operador de consulta individual en detalle.

Secuencia → Secuencia

La mayoría de los operadores de consulta caen en esta categoría, aceptando una o más secuencias como entrada y emitiendo una única secuencia de salida. La Figura 9-1 ilustra aquellos operadores que reestructuran la forma de las secuencias.

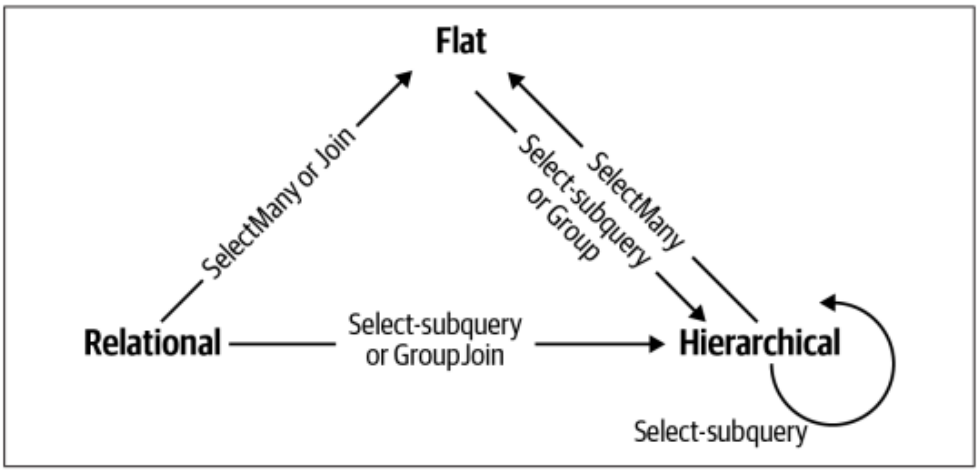


Figure 9-1. Shape-changing operators

Filtrado

```
IEnumerable<TSource> →IEnumerable<TSource>
```

Devuelve un subconjunto de los elementos originales.

```
Where, Take, TakeLast, TakeWhile, Skip, SkipLast, SkipWhile,
Distinct, DistinctBy
```

Proyección

```
IEnumerable<TSource>→IEnumerable<TResult>
```

Transforma cada elemento con una función lambda. SelectMany aplana secuencias anidadas; Select y SelectMany realizan uniones internas, uniones externas izquierdas, uniones cruzadas y uniones no equijoin con EF Core.

```
Select, SelectMany
```

Unión

```
IEnumerable<TOuter>, IEnumerable<TInner>→IEnumerable<TResult>
```

Combina elementos de una secuencia con otra. Los operadores Join y GroupJoin están diseñados para ser eficientes con consultas locales y admiten uniones internas y externas izquierdas. El operador Zip enumera dos secuencias al mismo tiempo, aplicando una función a cada par de elementos. En lugar de nombrar los argumentos de tipo TOuter y TInner, el operador Zip los nombra TFirst y TSecond:

```
IEnumerable<TFirst>, IEnumerable<TSecond>→IEnumerable<TResult>
```

```
Join, GroupJoin, Zip
```

Ordenamiento

```
IEnumerable<TSource>→IOrderedEnumerable<TSource>
```

Devuelve una secuencia reordenada.

```
OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
```

Agrupamiento

```
IEnumerable<TSource>→IEnumerable<IGrouping<TKey,TElement>>
```

```
IEnumerable<TSource>→IEnumerable<TElement[]>
```

Agrupa una secuencia en subsecuencias.

```
GroupBy, Chunk
```

Operadores de conjunto

```
IEnumerable<TSource>, IEnumerable<TSource>→IEnumerable<TSource>
```

Toma dos secuencias del mismo tipo y devuelve su intersección, unión o diferencia.

```
Concat, Union, UnionBy, Intersect, IntersectBy, Except, ExceptBy
```

Métodos de conversión: Importación

```
IEnumerable→IEnumerable<TResult>
```

```
OfType, Cast
```

Métodos de conversión: Exportación

```
IEnumerable<TSource>→An array, list, dictionary, lookup, or sequence
```

```
ToArray, ToList, ToDictionary, ToLookup, AsEnumerable, AsQueryable
```

Secuencia → Elemento o Valor

Los siguientes operadores de consulta aceptan una secuencia de entrada y emiten un solo elemento o valor.

Operadores de elementos

`IEnumerable<TSource>→TSource`

Selecciona un solo elemento de una secuencia.

```
First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault,
ElementAt, ElementAtOrDefault, MinBy, MaxBy, DefaultIfEmpty
```

Métodos de agregación

`IEnumerable<TSource>→scalar`

Realiza un cálculo en una secuencia, devolviendo un valor escalar (normalmente un número).

```
Aggregate, Average, Count, LongCount, Sum, Max, Min
```

Cuantificadores

`IEnumerable<TSource>→bool`

Un agregado que devuelve verdadero o falso.

```
All, Any, Contains, SequenceEqual
```

Void → Secuencia

En la tercera y última categoría se encuentran los operadores de consulta que generan una secuencia de salida desde cero.

Métodos de generación

`void→IEnumerable<TResult>`

Crea una secuencia simple.

```
Empty, Range, Repeat
```

Filtrado

`IEnumerable<TSource>→IEnumerable<TSource>`

Method	Description	SQL equivalents
`Where`	Returns a subset of elements that satisfy a given condition	`WHERE`
`Take`	Returns the first `count` elements and discards the rest	`WHERE ROW_NUMBER()...` *or* `TOP` *n* subquery
`Skip`	Ignores the first `count` elements and returns the rest	`WHERE ROW_NUMBER()...` *or* `NOT IN (SELECT TOP n...)`
`TakeLast`	Takes only the last `count` elements	Exception thrown
`SkipLast`	Takes only the last `count` elements	Exception thrown
`TakeWhile`	Emits elements from the input sequence until the predicate is false	Exception thrown

Method	Description	SQL equivalents
`SkipWhile`	Ignores elements from the input sequence until the predicate is false, and then emits the rest	Exception thrown
`Distinct`, `DistinctBy`	Returns a sequence that excludes duplicates	`SELECT DISTINCT...`



El column "Equivalente SQL" en las tablas de referencia de este capítulo no necesariamente corresponde a lo que una implementación IQueryable como EF Core producirá. Más bien, indica lo que normalmente usarías si estuvieras escribiendo la consulta SQL tú mismo. Donde no hay una traducción simple, la columna se deja en blanco. Donde no hay ninguna traducción en absoluto, la columna dice "Excepción lanzada".

El código de implementación Enumerable, cuando se muestra, excluye la verificación de argumentos nulos y predicados de indexación.

Con cada uno de los métodos de filtrado, siempre obtienes el mismo número o menos elementos que los que tenías al inicio. ¡Nunca puedes obtener más! Los elementos también son idénticos cuando salen; no se transforman de ninguna manera.

Where

Argument	Type
Source sequence	`IEnumerable<TSource>`
Predicate	`TSource => bool or (TSource,int) => bool`[a]

[a] Prohibited with LINQ to SQL and Entity Framework

Sintaxis de consulta

```
where bool-expression
```

Implementación Enumerable.Where

La implementación interna de Enumerable.Where, excluyendo la comprobación de nulos, es funcionalmente equivalente a lo siguiente:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func <TSource, bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

Resumen

Where devuelve los elementos de la secuencia de entrada que satisfacen el predicado dado. Por ejemplo:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query = names.Where (name => name.EndsWith ("y"));

// Harry
// Mary
// Jay
```

En la sintaxis de consulta:

```
IEnumerable<string> query = from n in names
                            where n.EndsWith ("y")
                            select n;
```

Una cláusula where puede aparecer más de una vez en una consulta y puede intercalarse con cláusulas let, orderby y join:

```
from n in names
where n.Length > 3
let u = n.ToUpper()
where u.EndsWith ("Y")
select u;

// HARRY
// MARY
```

Las reglas estándar de ámbito de C# se aplican a estas consultas. En otras palabras, no puedes hacer referencia a una variable antes de declararla con una variable de rango o una cláusula let.

Filtrado indexado

El predicado de Where opcionalmente acepta un segundo argumento, de tipo int. Esto se alimenta con la posición de cada elemento dentro de la secuencia de entrada, lo que permite que el predicado utilice esta información en su decisión de filtrado. Por ejemplo, el siguiente código salta cada segundo elemento:

```
IEnumerable<string> query = names.Where ((n, i) => i % 2 == 0);

// Tom
// Harry
// Jay
```

Se produce una excepción si utilizas el filtrado indexado en EF Core.

Comparaciones SQL LIKE en EF Core:

Los siguientes métodos en string se traducen al operador LIKE de SQL:

```
Contains, StartsWith, EndsWith
```

Por ejemplo, c.Name.Contains("abc") se traduce a customer.Name LIKE '%abc%' (o más precisamente, a una versión parametrizada de esto). Contains te permite comparar solo con una expresión evaluada localmente; para comparar con otra columna, debes usar el método EF.Functions.Like:

```
... where EF.Functions.Like (c.Description, "%" + c.Name + "%")
```

EF.Functions.Like también te permite realizar comparaciones más complejas (por ejemplo, LIKE 'abc%def%').

Comparaciones < and > de cadenas en EF Core

Puedes realizar comparaciones de orden en cadenas con el método CompareTo de string; esto se traduce a los operadores < y > de SQL:

```
dbContext.Purchases.Where (p => p.Description.CompareTo ("C") < 0)
```

WHERE x IN (..., ..., ...) en EF Core

Con EF Core, puedes aplicar el operador Contains a una colección local dentro de un predicado de filtro. Por ejemplo:

```
string[] chosenOnes = { "Tom", "Jay" };

from c in dbContext.Customers
where chosenOnes.Contains (c.Name)
...
```

Esto se traduce al operador IN de SQL. En otras palabras:

```
WHERE customer.Name IN ("Tom", "Jay")
```

Si la colección local es un array de entidades o tipos no escalares, EF Core podría en su lugar emitir una cláusula EXISTS.

Take, TakeLast, Skip y SkipLast

Argument	Type
Source sequence	IEnumerable<TSource>
Number of elements to take or skip	int

Take emite los primeros n elementos y descarta el resto; Skip descarta los primeros n elementos y emite el resto. Estos dos métodos son útiles juntos al implementar una página web que permite al usuario navegar a través de un gran conjunto de registros coincidentes. Por ejemplo, supongamos que un usuario busca en una base de datos de libros el término "mercury" y hay 100 coincidencias. Lo siguiente devuelve los primeros 20:

```
IQueryable<Book> query = dbContext.Books
  .Where   (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Take (20);
```

La siguiente consulta devuelve los libros del 21 al 40:

```
IQueryable<Book> query = dbContext.Books
  .Where   (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Skip (20).Take (20);
```

EF Core traduce Take y Skip a la función ROW_NUMBER en SQL Server 2005, o a una subconsulta TOP n en versiones anteriores de SQL Server.

Los métodos TakeLast y SkipLast toman o saltan los últimos n elementos.

A partir de .NET 6, el método Take tiene una sobrecarga que acepta una variable Range. Esta sobrecarga puede subsumir la funcionalidad de los cuatro métodos; por ejemplo, Take(5..) es equivalente a Skip(5), y Take(..^5) es equivalente a SkipLast(5).

TakeWhile y SkipWhile

Argument	Type
Source sequence	IEnumerable<TSource>
Predicate	TSource => bool or (TSource,int) => bool

TakeWhile enumera la secuencia de entrada, emitiendo cada elemento hasta que el predicado dado sea falso. Luego ignora los elementos restantes:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var takeWhileSmall = numbers.TakeWhile (n => n < 100);   // { 3, 5, 2 }
```

SkipWhile enumera la secuencia de entrada, ignorando cada elemento hasta que el predicado dado sea falso. Luego emite los elementos restantes:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var skipWhileSmall = numbers.SkipWhile (n => n < 100);   // { 234, 4, 1 }
```

TakeWhile y SkipWhile no tienen traducción a SQL y lanzan una excepción si se usan en una consulta de EF Core.

Distinct y DistinctBy

Distinct devuelve la secuencia de entrada sin duplicados. Opcionalmente, se puede pasar un comparador de igualdad personalizado. Lo siguiente devuelve las letras distintas en una cadena:

```
char[] distinctLetters = "HelloWorld".Distinct().ToArray();
string s = new string (distinctLetters);                    // HeloWrd
```

Podemos llamar a los métodos LINQ directamente en una cadena porque string implementa IEnumerable<char>.

El método DistinctBy se introdujo en .NET 6 y te permite especificar un selector de clave que se aplicará antes de realizar la comparación de igualdad. El resultado de la siguiente expresión es {1,2,3}:

```
new[] { 1.0, 1.1, 2.0, 2.1, 3.0, 3.1 }.DistinctBy (n => Math.Round (n, 0))
```

Proyección

`IEnumerable<TSource>→ IEnumerable<TResult>`

Method	Description	SQL equivalents
`Select`	Transforms each input element with the given lambda expression	`SELECT`
`SelectMany`	Transforms each input element, and then flattens and concatenates the resultant subsequences	`INNER JOIN`, `LEFT OUTER JOIN`, `CROSS JOIN`



Cuando se realiza una consulta en una base de datos, Select y SelectMany son las construcciones de unión más versátiles; para consultas locales, Join y GroupJoin son las construcciones de unión más eficientes.

Select

Argument	Type
Source sequence	`IEnumerable<TSource>`
Result selector	`TSource => TResult` or `(TSource,int) => TResult`[a]

[a] Prohibited with EF Core

Sintaxis de consulta

```
select projection-expression
```

Implementación en Enumerable

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

Descripción general

Con Select, siempre obtienes la misma cantidad de elementos con la que comenzaste. Sin embargo, cada elemento puede transformarse de cualquier manera mediante la función lambda. El siguiente ejemplo selecciona los nombres de todas las fuentes instaladas en la computadora (de System.Drawing):

```
IEnumerable<string> query = from f in FontFamily.Families
                            select f.Name;

foreach (string name in query) Console.WriteLine (name);
```

En este ejemplo, la cláusula select convierte un objeto FontFamily en su nombre. Aquí está el equivalente en lambda:

```
IEnumerable<string> query = FontFamily.Families.Select (f => f.Name);
```

Las declaraciones Select se utilizan a menudo para proyectar en tipos anónimos:

```
var query =
  from f in FontFamily.Families
  select new { f.Name, LineSpacing = f.GetLineSpacing (FontStyle.Bold) };
```

Una proyección sin transformación a veces se utiliza con la sintaxis de consulta para cumplir con el requisito de que la consulta termine en una cláusula select o group. El siguiente ejemplo selecciona las fuentes que admiten strikeout:

```
IEnumerable<FontFamily> query =
  from f in FontFamily.Families
  where f.IsStyleAvailable (FontStyle.Strikeout)
  select f;

foreach (FontFamily ff in query) Console.WriteLine (ff.Name);
```

En estos casos, el compilador omite la proyección al traducir a sintaxis fluida.

Proyección indexada

La expresión del selector puede aceptar opcionalmente un argumento entero, que actúa como un índice, proporcionando a la expresión la posición de cada elemento en la secuencia de entrada. Esto solo funciona con consultas locales:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query = names
  .Select ((s,i) => i + "=" + s);      //  { "0=Tom", "1=Dick", ... }
```

Subconsultas Select y jerarquías de objetos

Puedes anidar una subconsulta en una cláusula select para construir una jerarquía de objetos. El siguiente ejemplo devuelve una colección que describe cada directorio en Path.GetTempPath(), con una subcolección de archivos en cada directorio:

```
string tempPath = Path.GetTempPath();
DirectoryInfo[] dirs = new DirectoryInfo (tempPath).GetDirectories();

var query =
  from d in dirs
  where (d.Attributes & FileAttributes.System) == 0
  select new
  {
    DirectoryName = d.FullName,
    Created = d.CreationTime,

     Files = from f in d.GetFiles()
             where (f.Attributes & FileAttributes.Hidden) == 0
             select new { FileName = f.Name, f.Length, }
   };

 foreach (var dirFiles in query)
 {
   Console.WriteLine ("Directory: " + dirFiles.DirectoryName);
   foreach (var file in dirFiles.Files)
     Console.WriteLine ("  " + file.FileName + " Len: " + file.Length);
 }
```

La parte interna de esta consulta se puede llamar una subconsulta correlacionada. Una subconsulta está correlacionada si hace referencia a un objeto en la consulta externa, en este caso, hace referencia a "d", el directorio que se está enumerando.



Una subconsulta dentro de un Select te permite mapear una jerarquía de objetos a otra, o mapear un modelo de objeto relacional a un modelo de objeto jerárquico.

Con consultas locales, una subconsulta dentro de un Select provoca una ejecución doblemente diferida. En nuestro ejemplo, los archivos no se filtran ni se proyectan hasta que la instrucción foreach interna los enumera.

Subconsultas y joins en EF Core

Las subconsultas y los joins en EF Core funcionan bien. Puedes usar las subconsultas para realizar el trabajo de los joins al estilo SQL. Aquí te mostramos cómo obtener el nombre de cada cliente junto con sus compras de alto valor:

```
var query =
  from c in dbContext.Customers
  select new {
              c.Name,
              Purchases = (from p in dbContext.Purchases
                           where p.CustomerID == c.ID && p.Price > 1000
                           select new { p.Description, p.Price })
                           .ToList()
            };

foreach (var namePurchases in query)
{
  Console.WriteLine ("Customer: " + namePurchases.Name);
  foreach (var purchaseDetail in namePurchases.Purchases)
    Console.WriteLine ("  - $$$: " + purchaseDetail.Price);
}
```


Ten en cuenta el uso de ToList en la subconsulta. En EF Core 3, no es posible crear consultas a partir del resultado de la subconsulta cuando esa subconsulta hace referencia al DbContext. Este problema está siendo investigado por el equipo de EF Core y podría resolverse en una versión futura.


Este estilo de consulta es ideal para consultas interpretadas. La consulta externa y la subconsulta se procesan como una unidad, evitando viajes innecesarios a la base de datos. Sin embargo, en consultas locales, puede ser ineficiente porque se deben enumerar todas las combinaciones de elementos externos e internos para obtener las pocas combinaciones coincidentes. Una mejor opción para consultas locales es el uso de Join o GroupJoin, que se describen en las siguientes secciones.

Esta consulta combina objetos de dos colecciones diferentes y se puede considerar como un "join". La diferencia entre esto y un join (o subconsulta) convencional en una base de datos es que no estamos aplanando la salida en un único conjunto de resultados bidimensional. Estamos mapeando los datos relacionales a datos jerárquicos, en lugar de datos planos. Aquí está la misma consulta simplificada utilizando la propiedad de navegación "Purchases" en la entidad "Customer":

```
from c in dbContext.Customers
select new
{
  c.Name,
  Purchases = from p in c.Purchases    // Purchases is List<Purchase>
              where p.Price > 1000
              select new { p.Description, p.Price }
};
```

(En EF Core 3, no es necesario utilizar ToList al realizar la subconsulta en una propiedad de navegación).

Ambas consultas son análogas a un left outer join en SQL en el sentido de que obtenemos todos los clientes en la enumeración externa, independientemente de si tienen o no compras. Para emular un inner join, donde se excluyen los clientes sin compras de alto valor, necesitaríamos agregar una condición de filtro en la colección "Purchases":

```
from c in dbContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select new {
             c.Name,
             Purchases = from p in c.Purchases
                         where p.Price > 1000
                         select new { p.Description, p.Price }
           };
```

Sin embargo, esto es un poco confuso ya que hemos escrito el mismo predicado (Price > 1000) dos veces. Podemos evitar esta duplicación utilizando una cláusula "let":

```
from c in dbContext.Customers
let highValueP = from p in c.Purchases
                 where p.Price > 1000
                 select new { p.Description, p.Price }
where highValueP.Any()
select new { c.Name, Purchases = highValueP };
```

Este estilo de consulta es flexible. Al cambiar Any por Count, por ejemplo, podemos modificar la consulta para recuperar solo los clientes con al menos dos compras de alto valor:

```
...
where highValueP.Count() >= 2
select new { c.Name, Purchases = highValueP };
```

Proyectando en tipos concretos

En los ejemplos anteriores, hemos instanciado tipos anónimos en la salida. También puede ser útil instanciar clases con nombre (ordinarias), que se pueden poblar con inicializadores de objetos. Estas clases pueden incluir lógica personalizada y se pueden pasar entre métodos y ensamblados sin necesidad de utilizar información de tipo.

Un ejemplo típico es una entidad comercial personalizada. Una entidad comercial personalizada es simplemente una clase que escribes con algunas propiedades, pero está diseñada para ocultar detalles de nivel inferior relacionados con la base de datos. Por ejemplo, podrías excluir campos de clave externa de las clases de entidades comerciales. Suponiendo que hayamos escrito clases de entidad personalizadas llamadas CustomerEntity y PurchaseEntity, así es como podríamos proyectar en ellas:

```
IQueryable<CustomerEntity> query =
  from c in dbContext.Customers
  select new CustomerEntity
  {
    Name = c.Name,
    Purchases =
      (from p in c.Purchases
       where p.Price > 1000
       select new PurchaseEntity {
                                    Description = p.Description,
                                    Value = p.Price
                                  }
      ).ToList()
  };

// Force query execution, converting output to a more convenient List:
List<CustomerEntity> result = query.ToList();
```



Cuando se crean para transferir datos entre capas en un programa o entre sistemas separados, las clases de entidad comercial personalizadas a menudo se llaman Objetos de Transferencia de Datos (DTO, por sus siglas en inglés). Los DTO no contienen lógica empresarial.

Observa que hasta ahora no hemos tenido que usar una instrucción Join o SelectMany. Esto se debe a que estamos manteniendo la forma jerárquica de los datos, como se ilustra en la Figura 9-2. Con LINQ, a menudo puedes evitar el enfoque tradicional de SQL de aplanar las tablas en un conjunto de resultados bidimensional.

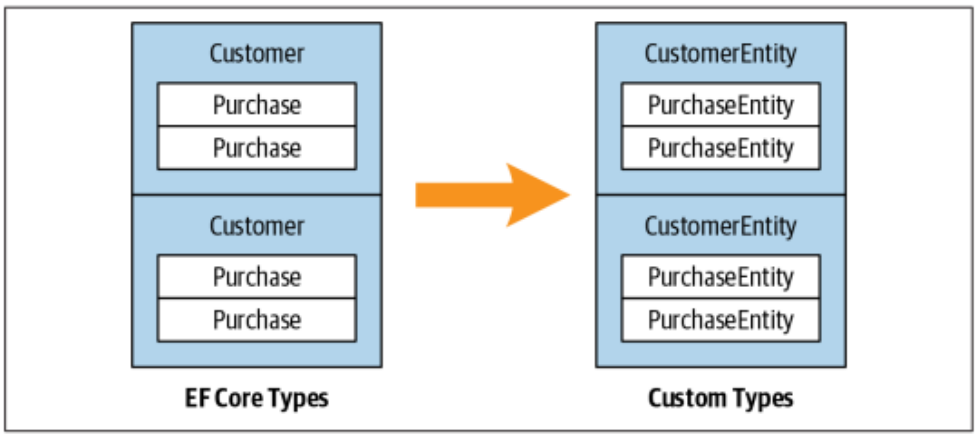


Figure 9-2. Projecting an object hierarchy

SelectMany

Argument	Type
Source sequence	`IEnumerable<TSource>`
Result selector	`TSource => IEnumerable<TResult>` or `(TSource,int) => IEnumerable<TResult>`[a]

[a] Prohibited with EF Core

Sintaxis de consulta (Query syntax)

```
from identifier1 in enumerable-expression1
from identifier2 in enumerable-expression2
...
```

Implementación Enumerable (Enumerable implementation)

```
public static IEnumerable<TResult> SelectMany<TSource,TResult>
  (IEnumerable<TSource> source,
   Func <TSource,IEnumerable<TResult>> selector)
{
  foreach (TSource element in source)
    foreach (TResult subElement in selector (element))
      yield return subElement;
}
```

Visión general

SelectMany concatena subsecuencias en una única secuencia de salida plana.

Recuerda que para cada elemento de entrada, Select produce exactamente un elemento de salida. En contraste, SelectMany produce de 0 a n elementos de salida. Los elementos de 0 a n provienen de una subsecuencia o secuencia hija que la expresión lambda debe emitir.

Puedes usar SelectMany para expandir secuencias hijas, aplanar colecciones anidadas y unir dos colecciones en una secuencia plana de salida. Usando la analogía de una cinta transportadora, SelectMany canaliza material fresco en una cinta transportadora. Con SelectMany, cada elemento de entrada es el desencadenante para la introducción de material fresco. El material fresco es emitido por la expresión lambda del selector y debe ser una secuencia. En otras palabras, la expresión lambda debe emitir una secuencia hija por cada elemento de entrada. El resultado final es una concatenación de las secuencias hijas emitidas para cada elemento de entrada.

Comenzando con un ejemplo simple, supongamos que tenemos el siguiente array de nombres:

```
string[] fullNames = { "Anne Williams", "John Fred Smith", "Sue Green" };
```

y queremos convertirlo en una única colección plana de palabras, es decir:

```
"Anne", "Williams", "John", "Fred", "Smith", "Sue", Green"
```

SelectMany es ideal para esta tarea, porque estamos mapeando cada elemento de entrada a un número variable de elementos de salida. Lo único que debemos hacer es crear una expresión de selección que convierta cada elemento de entrada en una secuencia hija. string.Split hace el trabajo muy bien: toma una cadena y la divide en palabras, emitiendo el resultado como un array:

```
string testInputElement = "Anne Williams";
string[] childSequence  = testInputElement.Split();

// childSequence is { "Anne", "Williams" };
```

Entonces, aquí está nuestra consulta SelectMany y el resultado:

```
IEnumerable<string> query = fullNames.SelectMany (name => name.Split());

foreach (string name in query)
  Console.Write (name + "|");  // Anne|Williams|John|Fred|Smith|Sue|Green|
```



Si reemplazas SelectMany con Select, obtienes los mismos resultados en forma jerárquica. Lo siguiente emite una secuencia de matrices de cadenas, requiriendo bucles foreach anidados para enumerar:

```
IEnumerable<string[]> query =
  fullNames.Select (name => name.Split());

foreach (string[] stringArray in query)
  foreach (string name in stringArray)
    Console.Write (name + "|");
```

La ventaja de SelectMany es que produce una única secuencia de resultado plana.

SelectMany es compatible con la sintaxis de consulta y se invoca al tener un generador adicional, es decir, una cláusula from adicional en la consulta. La palabra clave from tiene dos significados en la sintaxis de consulta. Al comienzo de una consulta, introduce la variable de rango original y la secuencia de entrada. En cualquier otro lugar de la consulta, se traduce como SelectMany. Aquí está nuestra consulta en sintaxis de consulta:

```
IEnumerable<string> query =
  from fullName in fullNames
  from name in fullName.Split()     // Translates to SelectMany
  select name;
```

Ten en cuenta que el generador adicional introduce una nueva variable de rango, en este caso, "name". Sin embargo, la antigua variable de rango permanece en ámbito y podemos acceder a ambas posteriormente.

Variables de rango múltiples

En el ejemplo anterior, tanto "name" como "fullName" permanecen en ámbito hasta que la consulta termine o alcance una cláusula "into". El alcance extendido de estas variables es el escenario clave para la sintaxis de consulta sobre la sintaxis fluida.

Para ilustrar esto, podemos tomar la consulta anterior e incluir "fullName" en la proyección final:

```
IEnumerable<string> query =
  from fullName in fullNames
  from name in fullName.Split()
  select name + " came from " + fullName;

Anne came from Anne Williams
Williams came from Anne Williams
John came from John Fred Smith
...
```

Detrás de escena, el compilador debe hacer algunos trucos para permitir el acceso a ambas variables. Una buena manera de apreciar esto es intentar escribir la misma consulta en sintaxis fluida. ¡Es complicado! Se vuelve aún más difícil si insertas una cláusula "where" o "orderby" antes de la proyección:

```
from fullName in fullNames
from name in fullName.Split()
orderby fullName, name
select name + " came from " + fullName;
```

El problema es que SelectMany emite una secuencia plana de elementos hijos, en nuestro caso, una colección plana de palabras. El elemento "externo" original del cual proviene (fullName) se pierde. La solución es "transportar" el elemento externo con cada hijo, en un tipo anónimo temporal:

```
from fullName in fullNames
from x in fullName.Split().Select (name => new { name, fullName } )
orderby x.fullName, x.name
select x.name + " came from " + x.fullName;
```

El único cambio aquí es que envolvemos cada elemento hijo (name) en un tipo anónimo que también contiene su fullName. Esto es similar a cómo se resuelve una cláusula "let". Aquí tienes la conversión final a sintaxis fluida:

```
IEnumerable<string> query = fullNames
  .SelectMany (fName => fName.Split()
                          .Select (name => new { name, fName } ))
  .OrderBy (x => x.fName)
  .ThenBy  (x => x.name)
  .Select  (x => x.name + " came from " + x.fName);
```

Pensando en la sintaxis de consulta

Como acabamos de demostrar, existen buenas razones para utilizar la sintaxis de consulta si necesitas múltiples variables de rango. En esos casos, ayuda no solo usar la sintaxis de consulta, sino también pensar directamente en sus términos.

Hay dos patrones básicos al escribir generadores adicionales. El primero es expandir y aplanar subsecuencias. Para hacer esto, llamas a una propiedad o método en una variable de rango existente en tu generador adicional. Hicimos esto en el ejemplo anterior:

```
from fullName in fullNames
from name in fullName.Split()
```

Aquí, hemos expandido desde la enumeración de nombres completos hasta la enumeración de palabras. Un ejemplo análogo en EF Core es cuando expandes propiedades de navegación de colecciones. La siguiente consulta lista todos los clientes junto con sus compras:

```
IEnumerable<string> query = from c in dbContext.Customers
                            from p in c.Purchases
                            select c.Name + " bought a " + p.Description;


Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
...
```

Aquí, hemos expandido cada cliente en una subsecuencia de compras.

El segundo patrón es realizar un producto cartesiano, o cruce, en el que cada elemento de una secuencia se combina con cada elemento de otra. Para hacer esto, introduce un generador cuya expresión de selección devuelve una secuencia no relacionada con una variable de rango:

```
int[] numbers = { 1, 2, 3 };  string[] letters = { "a", "b" };

IEnumerable<string> query = from n in numbers
                            from l in letters
                            select n.ToString() + l;

// RESULT: { "1a", "1b", "2a", "2b", "3a", "3b" }
```

Este estilo de consulta es la base de las uniones en estilo SelectMany.

Uniendo con SelectMany

Puedes usar SelectMany para unir dos secuencias simplemente filtrando los resultados de un producto cruzado. Por ejemplo, supongamos que queremos emparejar jugadores para un juego. Podríamos comenzar de la siguiente manera:

```
string[] players = { "Tom", "Jay", "Mary" };

IEnumerable<string> query = from name1 in players
                            from name2 in players
                            select name1 + " vs " + name2;

//RESULT: { "Tom vs Tom", "Tom vs Jay", "Tom vs Mary",
//          "Jay vs Tom", "Jay vs Jay", "Jay vs Mary",
//          "Mary vs Tom", "Mary vs "Jay", "Mary vs Mary" }
```

La consulta se lee como "Para cada jugador, iterar sobre cada jugador, seleccionando jugador 1 versus jugador 2". Aunque obtuvimos lo que pedimos (un producto cruzado), los resultados no son útiles hasta que agregamos un filtro:

```
IEnumerable<string> query = from name1 in players
                            from name2 in players
                            where name1.CompareTo (name2) < 0
                            orderby name1, name2
                            select name1 + " vs " + name2;

//RESULT: { "Jay vs Mary", "Jay vs Tom", "Mary vs Tom" }
```

El predicado del filtro constituye la condición de unión. Nuestra consulta puede llamarse unión no equitativa porque la condición de unión no utiliza un operador de igualdad.

SelectMany en EF Core

SelectMany en EF Core puede realizar productos cruzados, uniones no equitativas, uniones internas y uniones externas izquierdas. Puedes usar SelectMany tanto con asociaciones predefinidas como con relaciones ad hoc, al igual que con Select. La diferencia es que SelectMany devuelve un conjunto de resultados plano en lugar de jerárquico.

Un producto cruzado en EF Core se escribe de la misma manera que se mostró en la sección anterior. La siguiente consulta combina cada cliente con cada compra (un producto cruzado):

```
var query = from c in dbContext.Customers
            from p in dbContext.Purchases
            select c.Name + " might have bought a " + p.Description;
```

Sin embargo, lo más típico es que desees combinar clientes solo con sus propias compras. Esto se logra agregando una cláusula where con un predicado de unión. Esto resulta en una unión equi-join de estilo SQL estándar:

```
var query = from c in dbContext.Customers
            from p in dbContext.Purchases
            where c.ID == p.CustomerID
            select c.Name + " bought a " + p.Description;
```



Esto se traduce bien a SQL. En la próxima sección, veremos cómo se extiende para admitir uniones externas. Reformular estas consultas con el operador Join de LINQ en realidad las hace menos extensibles: LINQ es opuesto a SQL en este sentido.

Si tienes propiedades de navegación de colecciones en tus entidades, puedes expresar la misma consulta expandiendo la subcolección en lugar de filtrar el producto cruzado:

```
from c in dbContext.Customers
from p in c.Purchases
select new { c.Name, p.Description };
```

La ventaja es que hemos eliminado el predicado de unión. Hemos pasado de filtrar un producto cruzado a expandir y aplanar.

Puedes agregar cláusulas where a esta consulta para filtrar aún más. Por ejemplo, si solo queremos clientes cuyos nombres comiencen con "T", podríamos filtrar de la siguiente manera:

```
from c in dbContext.Customers
where c.Name.StartsWith ("T")
from p in c.Purchases
select new { c.Name, p.Description };
```

Esta consulta de EF Core funcionaría igual de bien si la cláusula where se moviera una línea hacia abajo, porque se genera el mismo SQL en ambos casos. Sin embargo, si es una consulta local, mover la cláusula where hacia abajo la haría menos eficiente. Con consultas locales, debes filtrar antes de hacer la unión.

Puedes introducir nuevas tablas en la mezcla con cláusulas adicionales from. Por ejemplo, si cada compra tuviera filas secundarias de elementos de compra, podrías producir un conjunto de

resultados plano de clientes con sus compras, cada una con sus líneas de detalle de compra de la siguiente manera:

```
from c in dbContext.Customers
from p in c.Purchases
from pi in p.PurchaseItems
select new { c.Name, p.Description, pi.Detail };
```

Cada cláusula from introduce una nueva tabla secundaria. Para incluir datos de una tabla principal (a través de una propiedad de navegación), no agregas una cláusula from, simplemente navegas hacia la propiedad. Por ejemplo, si cada cliente tiene un representante de ventas cuyo nombre quieres consultar, solo debes hacer esto:

```
from c in dbContext.Customers
select new { Name = c.Name, SalesPerson = c.SalesPerson.Name };
```

En este caso, no se utiliza SelectMany porque no hay una subcolección para aplanar. Las propiedades de navegación del padre devuelven un solo elemento.

Uniones externas con SelectMany

Vimos anteriormente que una subconsulta Select produce un resultado análogo a una unión externa izquierda:

```
from c in dbContext.Customers
select new {
             c.Name,
             Purchases = from p in c.Purchases
                         where p.Price > 1000
                         select new { p.Description, p.Price }
           };
```

En este ejemplo, se incluye cada elemento externo (cliente), sin importar si el cliente tiene compras o no. Pero supongamos que reescribimos esta consulta con SelectMany para obtener una única colección plana en lugar de un conjunto de resultados jerárquico:

```
from c in dbContext.Customers
from p in c.Purchases
where p.Price > 1000
select new { c.Name, p.Description, p.Price };
```

Al hacer esto y aplanar la consulta, hemos cambiado a una unión interna: ahora solo se incluyen los clientes para quienes existen una o más compras de alto valor. Para obtener una unión externa izquierda con un conjunto de resultados plano, debemos aplicar el operador de consulta DefaultIfEmpty a la secuencia interna. Este método devuelve una secuencia con un único elemento nulo si la secuencia de entrada no tiene elementos. Aquí está dicha consulta, sin tener en cuenta el predicado del precio:

```
from c in dbContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new { c.Name, p.Description, Price = (decimal?) p.Price };
```

Esto funciona perfectamente con EF Core, devolviendo todos los clientes, incluso si no tienen compras. Sin embargo, si ejecutáramos esto como una consulta local, daría error porque cuando

p es nulo, p.Description y p.Price generan una excepción de NullReferenceException. Podemos hacer que nuestra consulta sea robusta en ambos escenarios de la siguiente manera:

```
from c in dbContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new {
             c.Name,
             Descript = p == null ? null : p.Description,
             Price = p == null ? (decimal?) null : p.Price
           };
```

Ahora, volvamos a introducir el filtro de precio. No podemos usar una cláusula where como lo hicimos anteriormente, porque se ejecutaría después de DefaultIfEmpty:

```
from c in dbContext.Customers
from p in c.Purchases.DefaultIfEmpty()
where p.Price > 1000...
```

La solución correcta es insertar la cláusula Where antes de DefaultIfEmpty con una subconsulta:

```
from c in dbContext.Customers
from p in c.Purchases.Where (p => p.Price > 1000).DefaultIfEmpty()
select new {
             c.Name,
             Descript = p == null ? null : p.Description,
             Price = p == null ? (decimal?) null : p.Price
           };
```

EF Core traduce esto como una unión externa izquierda. Este es un patrón efectivo para escribir este tipo de consultas.



Si estás acostumbrado/a a escribir uniones externas en SQL, es posible que estés tentado/a a pasar por alto la opción más simple de una subconsulta de Select en favor del enfoque plano incómodo pero familiar centrado en SQL. El conjunto de resultados jerárquico de una subconsulta de Select suele ser más adecuado para consultas en estilo de unión externa, ya que no hay nulos adicionales con los que lidiar.

Joining

Method	Description	SQL equivalents
`Join`	Applies a lookup strategy to match elements from two collections, emitting a flat result set	`INNER JOIN`
`GroupJoin`	Similar to `Join`, but emits a *hierarchical* result set	`INNER JOIN`, `LEFT OUTER JOIN`
`Zip`	Enumerates two sequences in step (like a zipper), applying a function over each element pair	Exception thrown

Join y GroupJoin

```
IEnumerable<TOuter>, IEnumerable<TInner>→IEnumerable<TResult>
```

Join arguments

Argument	Type
Outer sequence	`IEnumerable<TOuter>`
Inner sequence	`IEnumerable<TInner>`
Outer key selector	`TOuter => TKey`
Inner key selector	`TInner => TKey`
Result selector	`(TOuter,TInner) => TResult`

Argumentos de GroupJoin

Argument	Type
Outer sequence	`IEnumerable<TOuter>`
Inner sequence	`IEnumerable<TInner>`
Outer key selector	`TOuter => TKey`
Inner key selector	`TInner => TKey`
Result selector	`(TOuter,IEnumerable<TInner>) => TResult`

Sintaxis de consulta (Query syntax)

```
from outer-var in outer-enumerable
join inner-var in inner-enumerable on outer-key-expr equals inner-key-expr
 [ into identifier ]
```

Visión general

Join y GroupJoin combinan dos secuencias de entrada en una única secuencia de salida. Join genera una salida plana (flat); GroupJoin genera una salida jerárquica (hierarchical).

Join y GroupJoin proporcionan una estrategia alternativa a Select y SelectMany. La ventaja de Join y GroupJoin es que se ejecutan de manera eficiente en colecciones locales en memoria porque primero cargan la secuencia interna en una búsqueda indexada por clave, evitando la necesidad de enumerar repetidamente cada elemento interno. La desventaja es que solo ofrecen equivalentes de inner join y left outer join; los cross joins y non-equi joins aún deben realizarse utilizando Select/SelectMany. En consultas de EF Core, Join y GroupJoin no ofrecen beneficios reales en comparación con Select y SelectMany.

La Tabla 9-1 resume las diferencias entre cada una de las estrategias de unión:

Table 9-1. Joining strategies

Strategy	Result shape	Local query efficiency	Inner joins	Left outer joins	Cross joins	Non-equi joins
`Select + SelectMany`	Flat	Bad	Yes	Yes	Yes	Yes
`Select + Select`	Nested	Bad	Yes	Yes	Yes	Yes
`Join`	Flat	Good	Yes	—	—	—
`GroupJoin`	Nested	Good	Yes	Yes	—	—
`GroupJoin + SelectMany`	Flat	Good	Yes	Yes	—	—

Join

El operador Join realiza una unión interna (inner join), generando una secuencia de salida plana (flat).

La siguiente consulta lista todos los clientes junto con sus compras sin utilizar una propiedad de navegación:

```
IQueryable<string> query =
  from c in dbContext.Customers
  join p in dbContext.Purchases on c.ID equals p.CustomerID
  select c.Name + " bought a " + p.Description;
```

Los resultados coinciden con lo que obtendríamos de una consulta en estilo SelectMany:

```
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
```

Para ver el beneficio de Join sobre SelectMany, debemos convertir esto en una consulta local. Podemos demostrar esto copiando todos los clientes y compras a matrices y luego consultando las matrices:

```
Customer[] customers = dbContext.Customers.ToArray();
Purchase[] purchases = dbContext.Purchases.ToArray();
var slowQuery = from c in customers
                from p in purchases where c.ID == p.CustomerID
                select c.Name + " bought a " + p.Description;

var fastQuery = from c in customers
                join p in purchases on c.ID equals p.CustomerID
                select c.Name + " bought a " + p.Description;
```

Aunque ambas consultas producen los mismos resultados, la consulta con Join es considerablemente más rápida porque su implementación en Enumerable pre-carga la colección interna (compras) en una búsqueda indexada por clave.

La sintaxis de la consulta para join se puede escribir en términos generales de la siguiente manera:

```
join inner-var in inner-sequence on outer-key-expr equals inner-key-expr
```

Los operadores de Join en LINQ diferencian entre la secuencia externa (outer sequence) y la secuencia interna (inner sequence).

Sintácticamente:

- La secuencia externa es la secuencia de entrada (clientes, en este caso).
- La secuencia interna es la nueva colección que introduces (compras, en este caso).

Join realiza uniones internas, lo que significa que los clientes sin compras se excluyen de la salida. Con las uniones internas, puedes intercambiar las secuencias interna y externa en la consulta y seguir obteniendo los mismos resultados:

```
from p in purchases                                 // p is now outer
join c in customers on p.CustomerID equals c.ID     // c is now inner
...
```

Puedes agregar más cláusulas de join a la misma consulta. Si cada compra, por ejemplo, tiene uno o más elementos de compra, podrías unir los elementos de compra de la siguiente manera:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID           // first join
join pi in purchaseItems on p.ID equals pi.PurchaseID     // second join
...
```

La variable "purchases" actúa como la secuencia interna en el primer join y como la secuencia externa en el segundo join. También podrías obtener los mismos resultados (ineficientemente) utilizando declaraciones anidadas de foreach de la siguiente manera:

```
foreach (Customer c in customers)
  foreach (Purchase p in purchases)
    if (c.ID == p.CustomerID)
      foreach (PurchaseItem pi in purchaseItems)
        if (p.ID == pi.PurchaseID)
          Console.WriteLine (c.Name + "," + p.Price + "," + pi.Detail);
```

En la sintaxis de consulta, las variables de los joins anteriores permanecen en alcance, al igual que ocurre con las consultas en estilo SelectMany. También se te permite insertar cláusulas where y let entre las cláusulas de join.

Unión en múltiples claves

Puedes unir en múltiples claves utilizando tipos anónimos de la siguiente manera:

```
from x in sequenceX
join y in sequenceY on new { K1 = x.Prop1, K2 = x.Prop2 }
               equals new { K1 = y.Prop3, K2 = y.Prop4 }
...
```

Para que esto funcione, los dos tipos anónimos deben tener una estructura idéntica. El compilador implementa ambos con el mismo tipo interno, lo que hace que las claves de unión sean compatibles.

Unión en sintaxis fluida

La siguiente consulta en sintaxis de consulta

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
select new { c.Name, p.Description, p.Price };
```

en sintaxis fluida se ve así:

```
customers.Join (                // outer collection
      purchases,                // inner collection
      c => c.ID,                // outer key selector
      p => p.CustomerID,        // inner key selector
      (c, p) => new
         { c.Name, p.Description, p.Price }    // result selector
);
```

El resultado de la expresión del selector de resultados al final crea cada elemento en la secuencia de salida. Si tienes cláusulas adicionales antes de proyectar, como orderby en este ejemplo:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
orderby p.Price
select c.Name + " bought a " + p.Description;
```

Debes crear un tipo anónimo temporal en el selector de resultados en sintaxis fluida. Esto mantiene tanto c como p en alcance después de la unión:

```
customers.Join (                  // outer collection
      purchases,                  // inner collection
      c => c.ID,                  // outer key selector
      p => p.CustomerID,          // inner key selector
      (c, p) => new { c, p } )    // result selector
  .OrderBy (x => x.p.Price)
  .Select  (x => x.c.Name + " bought a " + x.p.Description);
```

La sintaxis de consulta suele ser preferible al realizar uniones; es menos complicada.

GroupJoin

GroupJoin realiza el mismo trabajo que Join, pero en lugar de generar un resultado plano, genera un resultado jerárquico agrupado por cada elemento externo. También permite uniones externas izquierdas. Actualmente, GroupJoin no es compatible con EF Core.

La sintaxis de consulta para GroupJoin es la misma que para Join, pero se le agrega la palabra clave "into".

Aquí tienes el ejemplo más básico utilizando una consulta local:

```
Customer[] customers = dbContext.Customers.ToArray();
Purchase[] purchases = dbContext.Purchases.ToArray();

IEnumerable<IEnumerable<Purchase>> query =
  from c in customers
  join p in purchases on c.ID equals p.CustomerID
  into custPurchases
  select custPurchases;   // custPurchases is a sequence
```



La cláusula "into" se traduce como GroupJoin solo cuando aparece directamente después de una cláusula join. Después de una cláusula select o group, significa continuación de consulta. Los dos usos de la palabra clave "into" son bastante diferentes, aunque tienen una característica en común: ambos introducen una nueva variable de rango.

El resultado es una secuencia de secuencias, que se puede enumerar de la siguiente manera:

```
foreach (IEnumerable<Purchase> purchaseSequence in query)
  foreach (Purchase p in purchaseSequence)
    Console.WriteLine (p.Description);
```

Sin embargo, esto no es muy útil, ya que `purchaseSequence` no tiene referencia al cliente. Más comúnmente, harías lo siguiente:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
into custPurchases
select new { CustName = c.Name, custPurchases };
```

Esto produce los mismos resultados que la siguiente subconsulta Select (ineficiente):

```
from c in customers
select new
{
  CustName = c.Name,
  custPurchases = purchases.Where (p => c.ID == p.CustomerID)
};
```

Por defecto, `GroupJoin` realiza el equivalente a una unión externa izquierda. Para obtener una unión interna, donde los clientes sin compras se excluyen, debes filtrar en `custPurchases`:

```
from c in customers join p in purchases on c.ID equals p.CustomerID
into custPurchases
where custPurchases.Any()
select ...
```

Las cláusulas después de `into` en `group-join` operan en subsecuencias de elementos secundarios internos, no en elementos secundarios individuales. Esto significa que para filtrar compras individuales, necesitarías llamar a `Where` antes de hacer la unión:

```
from c in customers
join p in purchases.Where (p2 => p2.Price > 1000)
  on c.ID equals p.CustomerID
into custPurchases ...
```

Puedes construir consultas lambda con `GroupJoin` de la misma manera que lo harías con `Join`.

Uniones externas planas

Te encuentras en un dilema si deseas tanto una unión externa como un conjunto de resultados plano. `GroupJoin` te brinda la unión externa, `Join` te brinda el conjunto de resultados plano. La solución es llamar primero a `GroupJoin`, luego a `DefaultIfEmpty` en cada secuencia secundaria y, finalmente, a `SelectMany` en el resultado:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID into custPurchases
from cp in custPurchases.DefaultIfEmpty()
select new
{
  CustName = c.Name,
  Price = cp == null ? (decimal?) null : cp.Price
};
```

`DefaultIfEmpty` emite una secuencia con un solo valor nulo si una subsecuencia de compras está vacía. La segunda cláusula `from` se traduce a `SelectMany`. En este papel, el operador Join expande y aplanar todas las subsecuencias de compras, concatenándolas en una sola secuencia de elementos de compra.

Uniendo con lookups

Los métodos Join y GroupJoin en Enumerable funcionan en dos pasos. Primero, cargan la secuencia interna en un lookup. Segundo, consultan la secuencia externa en combinación con el lookup.

Un lookup es una secuencia de agrupaciones que se puede acceder directamente por clave. Otra forma de pensarlo es como un diccionario de secuencias, un diccionario que puede aceptar muchos elementos bajo cada clave (a veces llamado multidiccionario). Los lookups son de solo lectura y están definidos por la siguiente interfaz:

```
public interface ILookup<TKey,TElement> :
   IEnumerable<IGrouping<TKey,TElement>>, IEnumerable
{
  int Count { get; }
  bool Contains (TKey key);
  IEnumerable<TElement> this [TKey key] { get; }
}
```



Los operadores de unión, al igual que otros operadores que emiten secuencias, siguen la semántica de ejecución aplazada o perezosa. Esto significa que el lookup no se construye hasta que comienzas a enumerar la secuencia de salida (y luego se construye el lookup completo en ese momento).

Puedes crear y consultar lookups manualmente como una estrategia alternativa para usar los operadores de unión cuando trabajas con colecciones locales. Hay un par de beneficios al hacerlo:

- Puedes reutilizar el mismo lookup en múltiples consultas, así como en código imperativo ordinario.
- Consultar un lookup es una excelente manera de entender cómo funcionan Join y GroupJoin.

El método de extensión ToLookup crea un lookup. Lo siguiente carga todas las compras en un lookup, usando CustomerID como clave:

```
ILookup<int?,Purchase> purchLookup =
  purchases.ToLookup (p => p.CustomerID, p => p);
```

El primer argumento selecciona la clave, y el segundo argumento selecciona los objetos que se cargarán como valores en el lookup.

Leer un lookup es similar a leer un diccionario, excepto que el indexador devuelve una secuencia de elementos coincidentes en lugar de un solo elemento coincidente. Lo siguiente enumera todas las compras realizadas por el cliente cuyo ID es 1:

```
foreach (Purchase p in purchLookup [1])
  Console.WriteLine (p.Description);
```

Con un lookup en su lugar, puedes escribir consultas SelectMany/Select que se ejecuten tan eficientemente como las consultas Join/GroupJoin. Join es equivalente a usar SelectMany en un lookup:

```
from c in customers
from p in purchLookup [c.ID]
select new { c.Name, p.Description, p.Price };

Tom Bike 500
Tom Holiday 2000
Dick Bike 600
Dick Phone 300
...
```

Agregar una llamada a DefaultIfEmpty convierte esto en un outer join:

```
from c in customers
from p in purchLookup [c.ID].DefaultIfEmpty()
 select new {
              c.Name,
              Descript = p == null ? null : p.Description,
              Price = p == null ? (decimal?) null : p.Price
            };
```

GroupJoin es equivalente a leer el lookup dentro de una proyección:

```
from c in customers
select new {
             CustName = c.Name,
             CustPurchases = purchLookup [c.ID]
           };
```

Implementaciones de Enumerable

Aquí tienes la implementación más simple y válida de Enumerable.Join, sin tener en cuenta las comprobaciones de nulos:

```
public static IEnumerable <TResult> Join
                                    <TOuter,TInner,TKey,TResult> (
  this IEnumerable <TOuter>     outer,
  IEnumerable <TInner>          inner,
  Func <TOuter,TKey>            outerKeySelector,
  Func <TInner,TKey>            innerKeySelector,
  Func <TOuter,TInner,TResult>  resultSelector)
{
  ILookup <TKey, TInner> lookup = inner.ToLookup (innerKeySelector);
  return
    from outerItem in outer
    from innerItem in lookup [outerKeySelector (outerItem)]
    select resultSelector (outerItem, innerItem);
}
```

La implementación de GroupJoin es similar a la de Join pero más simple:

```
public static IEnumerable <TResult> GroupJoin
                                    <TOuter,TInner,TKey,TResult> (
  this IEnumerable <TOuter>     outer,
  IEnumerable <TInner>          inner,
  Func <TOuter,TKey>            outerKeySelector,
  Func <TInner,TKey>            innerKeySelector,
  Func <TOuter,IEnumerable<TInner>,TResult>  resultSelector)
{
  ILookup <TKey, TInner> lookup = inner.ToLookup (innerKeySelector);
  return
    from outerItem in outer
    select resultSelector
     (outerItem, lookup [outerKeySelector (outerItem)]);
}
```

El operador Zip

```
IEnumerable<TFirst>, IEnumerable<TSecond>→IEnumerable<TResult>
```

El operador Zip enumera dos secuencias en paralelo (como un cierre relámpago), devolviendo una secuencia basada en la aplicación de una función a cada par de elementos. Por ejemplo, lo siguiente:

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip (words, (n, w) => n + "=" + w);
```

produce una secuencia con los siguientes elementos:

```
3=three
5=five
7=seven
```

Los elementos adicionales en cualquiera de las secuencias de entrada son ignorados. Zip no es compatible con EF Core.

Ordering

`IEnumerable<TSource>→IOrderedEnumerable<TSource>`

Method	Description	SQL equivalents
`OrderBy, ThenBy`	Sorts a sequence in ascending order	`ORDER BY ...`
`OrderByDescending, ThenByDescending`	Sorts a sequence in descending order	`ORDER BY ... DESC`
`Reverse`	Returns a sequence in reverse order	Exception thrown

Los operadores de ordenamiento devuelven los mismos elementos en un orden diferente.

LINQOrderBy, OrderByDescending, ThenBy y ThenByDescending

Argumentos de OrderBy y OrderByDescending

Argument	Type
Input sequence	`IEnumerable<TSource>`
Key selector	`TSource => TKey`

Return type = `IOrderedEnumerable<TSource>`

Argumentos de ThenBy y ThenByDescending

Argument	Type
Input sequence	`IOrderedEnumerable<TSource>`
Key selector	`TSource => TKey`

Sintaxis de consulta

```
orderby expression1 [descending] [, expression2 [descending] ... ]
```

Resumen

OrderBy devuelve una versión ordenada de la secuencia de entrada, utilizando la expresión keySelector para hacer comparaciones. La siguiente consulta emite una secuencia de nombres en orden alfabético:

```
IEnumerable<string> query = names.OrderBy (s => s);
```

La siguiente ordena los nombres por longitud:

```
IEnumerable<string> query = names.OrderBy (s => s.Length);

// Result: { "Jay", "Tom", "Mary", "Dick", "Harry" };
```

El orden relativo de los elementos con la misma clave de ordenamiento (en este caso, Jay/Tom y Mary/Dick) es indeterminado, a menos que agregues un operador ThenBy:

```
IEnumerable<string> query = names.OrderBy (s => s.Length).ThenBy (s => s);

// Result: { "Jay", "Tom", "Dick", "Mary", "Harry" };
```

ThenBy reordena solo los elementos que tenían la misma clave de ordenamiento en la clasificación anterior. Puedes encadenar cualquier cantidad de operadores ThenBy. La siguiente clasifica primero por longitud, luego por el segundo carácter y finalmente por el primer carácter:

```
names.OrderBy (s => s.Length).ThenBy (s => s[1]).ThenBy (s => s[0]);
```

Aquí está el equivalente en sintaxis de consulta:

```
from s in names
orderby s.Length, s[1], s[0]
select s;
```



La siguiente variación es incorrecta: en realidad, ordenará primero por s[1] y luego por s.Length (o en el caso de una consulta a una base de datos, solo ordenará por s[1] y descartará el ordenamiento anterior):

```
from s in names
orderby s.Length
orderby s[1]
...
```

LINQ también proporciona los operadores OrderByDescending y ThenByDescending, que hacen lo mismo pero emiten los resultados en orden inverso. La siguiente consulta de EF Core recupera las compras en orden descendente de precio, con aquellas del mismo precio listadas alfabéticamente:

```
dbContext.Purchases.OrderByDescending (p => p.Price)
                   .ThenBy (p => p.Description);
```

En sintaxis de consulta:

```
from p in dbContext.Purchases
orderby p.Price descending, p.Description
select p;
```

Comparadores y collations

En una consulta local, los objetos de selección de clave determinan el algoritmo de ordenamiento a través de su implementación predeterminada de IComparable (ver Capítulo 7). Puedes anular el algoritmo de ordenamiento pasando un objeto IComparer. Lo siguiente realiza una ordenación sin distinción entre mayúsculas y minúsculas:

```
names.OrderBy (n => n, StringComparer.CurrentCultureIgnoreCase);
```

Pasar un comparador no es compatible con la sintaxis de consulta ni de ninguna manera con EF Core. Al consultar una base de datos, el algoritmo de comparación está determinado por la intercalación de la columna participante. Si la intercalación distingue entre mayúsculas y minúsculas, puedes solicitar una ordenación sin distinción entre mayúsculas y minúsculas llamando a ToUpper en el selector de clave:

```
from p in dbContext.Purchases
orderby p.Description.ToUpper()
select p;
```

IOrderedEnumerable e IOrderedQueryable

Los operadores de ordenamiento devuelven subtipos especiales de IEnumerable<T>. Los de Enumerable devuelven IOrderedEnumerable<TSource>, mientras que los de Queryable devuelven IOrderedQueryable<TSource>. Estos subtipos permiten que un operador ThenBy posterior refine en lugar de reemplazar el ordenamiento existente.

Los miembros adicionales que estos subtipos definen no se exponen públicamente, por lo que se presentan como secuencias ordinarias. El hecho de que sean tipos diferentes entra en juego al construir consultas de forma progresiva:

```
IOrderedEnumerable<string> query1 = names.OrderBy (s => s.Length);
IOrderedEnumerable<string> query2 = query1.ThenBy (s => s);
```

Si en cambio declaramos `query1` como tipo `IEnumerable<string>`, la segunda línea no se compilaría, ya que ThenBy requiere una entrada de tipo `IOrderedEnumerable<string>`. Puedes evitar preocuparte por esto al usar la inferencia de tipos implícita para las variables de rango:

```
var query1 = names.OrderBy (s => s.Length);
var query2 = query1.ThenBy (s => s);
```

Sin embargo, la inferencia de tipos implícita también puede generar problemas. Lo siguiente no se compilará:

```
var query = names.OrderBy (s => s.Length);
query = query.Where (n => n.Length > 3);       // Compile-time error
```

El compilador infiere que `query` es de tipo `IOrderedEnumerable<string>`, según el tipo de secuencia de salida de `OrderBy`. Sin embargo, el `Where` en la línea siguiente devuelve un `IEnumerable<string>` ordinario, que no se puede asignar de nuevo a `query`. Puedes solucionar esto tanto con tipado explícito como llamando a `AsEnumerable()` después de `OrderBy`:

```
var query = names.OrderBy (s => s.Length).AsEnumerable();
query = query.Where (n => n.Length > 3);                    // OK
```

El equivalente en consultas interpretadas es llamar a `AsQueryable`.

Grouping

Method	Description	SQL equivalents
GroupBy	Groups a sequence into subsequences	GROUP BY
Chunk	Groups a sequence into arrays of a fixed size	

GroupBy

```
IEnumerable<TSource>→IEnumerable<IGrouping<TKey,TElement>>
```

Argument	Type
Input sequence	IEnumerable<TSource>
Key selector	TSource => TKey
Element selector (optional)	TSource => TElement
Comparer (optional)	IEqualityComparer<TKey>

Sintaxis de consulta

```
group element-expression by key-expression
```

Resumen

GroupBy organiza una secuencia plana de elementos en secuencias de grupos. Por ejemplo, lo siguiente organiza todos los archivos en Path.GetTempPath() por extensión:

```
string[] files = Directory.GetFiles (Path.GetTempPath());

IEnumerable<IGrouping<string,string>> query =
  files.GroupBy (file => Path.GetExtension (file));
```

O, con tipado implícito:

```
var query = files.GroupBy (file => Path.GetExtension (file));
```

Aquí tienes cómo enumerar el resultado:

```
foreach (IGrouping<string,string> grouping in query)
{
  Console.WriteLine ("Extension: " + grouping.Key);
  foreach (string filename in grouping)
    Console.WriteLine ("   - " + filename);
}

Extension: .pdf
  -- chapter03.pdf
  -- chapter04.pdf
Extension: .doc
  -- todo.doc
  -- menu.doc
  -- Copy of menu.doc
...
```

Enumerable.GroupBy funciona leyendo los elementos de entrada en un diccionario temporal de listas para que todos los elementos con la misma clave terminen en la misma sublista. Luego emite una secuencia de grupos. Un grupo es una secuencia con una propiedad Key:

```
public interface IGrouping <TKey,TElement> : IEnumerable<TElement>,
                                             IEnumerable
{
  TKey Key { get; }    // Key applies to the subsequence as a whole
}
```

De forma predeterminada, los elementos en cada grupo son los elementos de entrada sin transformar, a menos que especifiques un argumento elementSelector. Lo siguiente proyecta cada elemento de entrada en mayúsculas:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper());
```

Un elementSelector es independiente del keySelector. En nuestro caso, esto significa que la clave en cada grupo todavía está en su caso original:

```
Extension: .pdf
  -- CHAPTER03.PDF
  -- CHAPTER04.PDF
Extension: .doc
  -- TODO.DOC
```

Ten en cuenta que las subcolecciones no se emiten en orden alfabético de la clave. GroupBy simplemente agrupa; no realiza ninguna clasificación. De hecho, preserva el orden original. Para ordenar, debes agregar un operador OrderBy:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper())
     .OrderBy (grouping => grouping.Key);
```

GroupBy tiene una traducción simple y directa en la sintaxis de consulta:

```
group element-expr by key-expr
```

Aquí está nuestro ejemplo en sintaxis de consulta:

```
from file in files
group file.ToUpper() by Path.GetExtension (file);
```

Al igual que con select, group "finaliza" una consulta, a menos que agregues una cláusula de continuación de consulta:

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
orderby grouping.Key
select grouping;
```

Las continuaciones de consulta a menudo son útiles en una consulta con group by. La siguiente consulta filtra los grupos que tienen menos de cinco archivos:

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
where grouping.Count() >= 5
select grouping;
```



Un where después de un group by es equivalente a HAVING en SQL. Se aplica a cada subsecuencia o grupo en su totalidad en lugar de a los elementos individuales.

A veces, solo estás interesado en el resultado de una agregación en un grupo y puedes abandonar las subsecuencias:

```
string[] votes = { "Dogs", "Cats", "Cats", "Dogs", "Dogs" };

IEnumerable<string> query = from vote in votes
                            group vote by vote into g
                            orderby g.Count() descending
                            select g.Key;

string winner = query.First();    // Dogs
```

Grouping en EF Core

El agrupamiento funciona de la misma manera al consultar una base de datos. Sin embargo, si tienes propiedades de navegación configuradas, descubrirás que la necesidad de agrupar surge con menos frecuencia que con SQL estándar. Por ejemplo, para seleccionar clientes con al menos dos compras, no es necesario agrupar; la siguiente consulta hace el trabajo correctamente:

```
from c in dbContext.Customers
where c.Purchases.Count >= 2
select c.Name + " has made " + c.Purchases.Count + " purchases";
```

Un ejemplo de cuándo podrías usar el agrupamiento es para listar las ventas totales por año:

```
from p in dbContext.Purchases
group p.Price by p.Date.Year into salesByYear
select new {
             Year       = salesByYear.Key,
             TotalValue = salesByYear.Sum()
           };
```

El agrupamiento en LINQ es más potente que el GROUP BY de SQL en el sentido de que puedes obtener todas las filas de detalle sin realizar ninguna agregación:

```
from p in dbContext.Purchases
group p by p.Date.Year
Date.Year
```

Sin embargo, esto no funciona en EF Core. Una solución fácil es llamar a .AsEnumerable() justo antes del agrupamiento para que el agrupamiento ocurra en el cliente. Esto no es menos eficiente siempre y cuando realices cualquier filtrado antes del agrupamiento para que solo obtengas los datos que necesitas del servidor.

Otra diferencia con el SQL tradicional es que no hay obligación de proyectar las variables o expresiones utilizadas en el agrupamiento o la clasificación.

Agrupación por múltiples claves

Puedes agrupar por una clave compuesta utilizando un tipo anónimo:

```
from n in names
group n by new { FirstLetter = n[0], Length = n.Length };
```

Comparadores de igualdad personalizados

Puedes pasar un comparador de igualdad personalizado a GroupBy, en una consulta local, para cambiar el algoritmo de comparación de claves. Sin embargo, rara vez se requiere esto, ya que cambiar la expresión del selector de claves generalmente es suficiente. Por ejemplo, lo siguiente crea un agrupamiento insensible a mayúsculas y minúsculas:

```
group n by n.ToUpper()
```

Chunk

```
IEnumerable<TSource>→IEnumerable<TElement[]>
```

Argument	Type
Input sequence	IEnumerable<TSource>
size	int

Introducido en .NET 6, el método "Chunk" agrupa una secuencia en fragmentos del tamaño especificado (o menos, si no hay suficientes elementos):

```
foreach (int[] chunk in new[] { 1, 2, 3, 4, 5, 6, 7, 8 }.Chunk (3))
  Console.WriteLine (string.Join (", ", chunk));
```

Output:

```
1, 2, 3
4, 5, 6
7, 8
```

Operadores Set (Set Operators)

`IEnumerable<TSource>, IEnumerable<TSource>→IEnumerable<TSource>`

Method	Description	SQL equivalents
`Concat`	Returns a concatenation of elements in each of the two sequences	`UNION ALL`
`Union, UnionBy`	Returns a concatenation of elements in each of the two sequences, excluding duplicates	`UNION`
`Intersect, IntersectBy`	Returns elements present in both sequences	`WHERE ... IN (...)`
`Except, ExceptBy`	Returns elements present in the first but not the second sequence	`EXCEPT` *or* `WHERE ... NOT IN (...)`

Concat, Union y UnionBy

Concat retorna todos los elementos de la primera secuencia, seguidos por todos los elementos de la segunda. Union hace lo mismo, pero elimina cualquier duplicado:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };

IEnumerable<int>
  concat = seq1.Concat (seq2),    //  { 1, 2, 3, 3, 4, 5 }
  union  = seq1.Union  (seq2);    //  { 1, 2, 3, 4, 5 }
```

Especificar el argumento de tipo explícitamente es útil cuando las secuencias tienen tipos diferentes pero los elementos tienen un tipo base común. Por ejemplo, con la API de reflexión (Capítulo 18), los métodos y propiedades se representan con las clases MethodInfo y PropertyInfo, que tienen una clase base común llamada MemberInfo. Podemos concatenar métodos y propiedades indicando explícitamente esa clase base al llamar a Concat:

```
MethodInfo[] methods = typeof (string).GetMethods();
PropertyInfo[] props = typeof (string).GetProperties();
IEnumerable<MemberInfo> both = methods.Concat<MemberInfo> (props);
```

En el siguiente ejemplo, filtramos los métodos antes de concatenar:

```
var methods = typeof (string).GetMethods().Where (m => !m.IsSpecialName);
var props = typeof (string).GetProperties();
var both = methods.Concat<MemberInfo> (props);
```

Este ejemplo se basa en la covarianza de los parámetros de tipo de la interfaz: methods es del tipo IEnumerable<MethodInfo>, que requiere una conversión covariante a IEnumerable<MemberInfo>. Es una buena ilustración de cómo la covarianza hace que las cosas funcionen de la manera que esperas.

UnionBy (introducido en .NET 6) toma un keySelector, que se utiliza para determinar si un elemento es duplicado. En el siguiente ejemplo, realizamos una unión sin distinguir mayúsculas y minúsculas:

```
string[] seq1 = { "A", "b", "C" };
string[] seq2 = { "a", "B", "c" };
var union = seq1.UnionBy (seq2, x => x.ToUpperInvariant());
// union is { "A", "b", "C" }
```

En este caso, se puede lograr lo mismo con Union si proporcionamos un comparador de igualdad:

```
var union = seq1.Union (seq2, StringComparer.InvariantCultureIgnoreCase);
```

Intersect, IntersectBy, Except y ExceptBy

Intersect devuelve los elementos que dos secuencias tienen en común. Except devuelve los elementos de la primera secuencia que no están presentes en la segunda:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };

IEnumerable<int>
  commonality = seq1.Intersect (seq2),    //  { 3 }
  difference1 = seq1.Except    (seq2),    //  { 1, 2 }
  difference2 = seq2.Except    (seq1);    //  { 4, 5 }
```

Enumerable.Except funciona internamente cargando todos los elementos de la primera colección en un diccionario y luego eliminando del diccionario todos los elementos presentes en la segunda secuencia. El equivalente en SQL es una subconsulta NOT EXISTS o NOT IN:

```
SELECT number FROM numbers1Table
WHERE number NOT IN (SELECT number FROM numbers2Table)
```

Los métodos IntersectBy y ExceptBy (de .NET 6) te permiten especificar un selector de clave que se aplica antes de realizar la comparación de igualdad (consulta la discusión sobre UnionBy en la sección anterior).

Métodos de conversión

LINQ se ocupa principalmente de secuencias, es decir, colecciones del tipo IEnumerable<T>. Los métodos de conversión permiten convertir hacia y desde otros tipos de colecciones:

Method	Description
`OfType`	Converts `IEnumerable` to `IEnumerable<T>`, discarding wrongly typed elements
`Cast`	Converts `IEnumerable` to `IEnumerable<T>`, throwing an exception if there are any wrongly typed elements
`ToArray`	Converts `IEnumerable<T>` to `T[]`
`ToList`	Converts `IEnumerable<T>` to `List<T>`
`ToDictionary`	Converts `IEnumerable<T>` to `Dictionary<TKey,TValue>`
`ToLookup`	Converts `IEnumerable<T>` to `ILookup<TKey,TElement>`
`AsEnumerable`	Upcasts to `IEnumerable<T>`
`AsQueryable`	Casts or converts to `IQueryable<T>`

OfType y Cast

OfType y Cast aceptan una colección IEnumerable no genérica y generan una secuencia IEnumerable<T> genérica que puedes consultar posteriormente:

```
ArrayList classicList = new ArrayList();          // in System.Collections
classicList.AddRange ( new int[] { 3, 4, 5 } );
IEnumerable<int> sequence1 = classicList.Cast<int>();
```

Cast y OfType difieren en su comportamiento al encontrar un elemento de entrada que tenga un tipo incompatible. Cast lanza una excepción, mientras que OfType ignora el elemento incompatible. Continuando con el ejemplo anterior:

```
DateTime offender = DateTime.Now;
classicList.Add (offender);
IEnumerable<int>
  sequence2 = classicList.OfType<int>(), // OK - ignores offending DateTime
  sequence3 = classicList.Cast<int>();   // Throws exception
```

Las reglas de compatibilidad de elementos siguen exactamente las del operador "is" de C#, por lo tanto, solo se consideran las conversiones de referencia y las conversiones de unboxing. Esto se puede observar examinando la implementación interna de OfType:

```
public static IEnumerable<TSource> OfType <TSource> (IEnumerable source)
{
  foreach (object element in source)
    if (element is TSource)

      yield return (TSource)element;
}
```

Cast tiene una implementación idéntica, excepto que omite la prueba de compatibilidad de tipos:

```
public static IEnumerable<TSource> Cast <TSource> (IEnumerable source)
{
  foreach (object element in source)
    yield return (TSource)element;
}
```

Una consecuencia de estas implementaciones es que no puedes usar Cast para realizar conversiones numéricas o personalizadas (para eso, debes realizar una operación Select en su lugar). En otras palabras, Cast no es tan flexible como el operador de conversión de C#:

```
int i = 3;
long l = i;         // Implicit numeric conversion int->long
int i2 = (int) l;   // Explicit numeric conversion long->int
```

Podemos demostrar esto al intentar usar OfType o Cast para convertir una secuencia de enteros a una secuencia de longs:

```
int[] integers = { 1, 2, 3 };

IEnumerable<long> test1 = integers.OfType<long>();
IEnumerable<long> test2 = integers.Cast<long>();
```

Cuando se enumeran, test1 emite cero elementos y test2 lanza una excepción.

Examinando la implementación de OfType, es bastante claro por qué. Después de sustituir TSource, obtenemos la siguiente expresión:

```
(element is long)
```

Esto devuelve false para un elemento de tipo int debido a la falta de una relación de herencia.



La razón por la que test2 lanza una excepción al enumerar es más sutil. Observa en la implementación de Cast que element es de tipo object. Cuando TSource es un tipo de valor, el CLR asume que se trata de una conversión de unboxing y sintetiza un método que reproduce el escenario descrito en la sección "Boxing and Unboxing" en la página 126:

```
int value = 123;
object element = value;
long result = (long) element;  // exception
```

Debido a que la variable element se declara de tipo object, se realiza una conversión de objeto a long (unboxing) en lugar de una conversión numérica de int a long. Las operaciones de unboxing requieren una coincidencia exacta de tipos, por lo que la operación de unboxing de objeto a long falla cuando se le proporciona un int.

Como sugerimos anteriormente, la solución es utilizar un Select ordinario:

```
IEnumerable<long> castLong = integers.Select (s => (long) s);
```

OfType y Cast también son útiles para realizar downcasting de elementos en una secuencia genérica de entrada. Por ejemplo, si tienes una secuencia de entrada de tipo IEnumerable<Fruit>, OfType<Apple> devolvería solo las manzanas. Esto es particularmente útil en LINQ to XML (ver Capítulo 10).

Cast tiene soporte para sintaxis de consulta: simplemente antecede la variable de rango con un tipo:

```
from TreeNode node in myTreeView.Nodes
...
```

ToArray, ToList, ToDictionary, ToHashSet y ToLookup

ToArray, ToList y ToHashSet emiten los resultados en un array, List<T> o HashSet<T>. Cuando se ejecutan, estos operadores fuerzan la enumeración inmediata de la secuencia de entrada. Para ejemplos, consulta "Ejecución Diferida" en la página 408 del Capítulo 8.

ToDictionary y ToLookup aceptan los siguientes argumentos:

Argument	Type
Input sequence	`IEnumerable<TSource>`
Key selector	`TSource => TKey`
Element selector (optional)	`TSource => TElement`
Comparer (optional)	`IEqualityComparer<TKey>`

ToDictionary también fuerza la ejecución inmediata de una secuencia, escribiendo los resultados en un Dictionary genérico. La expresión keySelector que proporciones debe evaluar a un valor único para cada elemento en la secuencia de entrada; de lo contrario, se lanzará una excepción. En cambio, ToLookup permite tener muchos elementos con la misma clave. Describimos los lookups en "Joining with lookups" en la página 474.

AsEnumerable y AsQueryable

AsEnumerable realiza un upcast de una secuencia a IEnumerable<T>, lo que obliga al compilador a vincular los operadores de consulta posteriores a los métodos en Enumerable en lugar de Queryable. Para un ejemplo, consulta "Combinando consultas interpretadas y locales" en la página 428.

AsQueryable realiza un downcast de una secuencia a IQueryable<T> si implementa esa interfaz. De lo contrario, instancia un envoltorio IQueryable<T> sobre la consulta local.

Operadores de Elementos (Element Operators)

`IEnumerable<TSource>→ TSource`

Method	Description	SQL equivalents
`First, FirstOrDefault`	Returns the first element in the sequence, optionally satisfying a predicate	`SELECT TOP 1 ... ORDER BY ...`
`Last, LastOrDefault`	Returns the last element in the sequence, optionally satisfying a predicate	`SELECT TOP 1 ... ORDER BY ... DESC`
`Single, SingleOrDefault`	Equivalent to `First/FirstOrDefault`, but throws an exception if there is more than one match	
`ElementAt, ElementAtOrDefault`	Returns the element at the specified position	Exception thrown
`MinBy, MaxBy`	Returns the element with the smallest or largest value	Exception thrown
`DefaultIfEmpty`	Returns a single-element sequence whose value is `default(TSource)` if the sequence has no elements	`OUTER JOIN`

Los métodos que terminan en "OrDefault" devuelven default(TSource) en lugar de lanzar una excepción si la secuencia de entrada está vacía o si no hay elementos que coincidan con el predicado proporcionado.

default(TSource) es null para elementos de tipo referencia, false para el tipo bool y cero para tipos numéricos.

First, Last y Single

Argument	Type
Source sequence	`IEnumerable<TSource>`
Predicate (optional)	`TSource => bool`

El siguiente ejemplo muestra el uso de First y Last:

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int first      = numbers.First();                      // 1
int last       = numbers.Last();                       // 5
int firstEven  = numbers.First  (n => n % 2 == 0);     // 2
int lastEven   = numbers.Last   (n => n % 2 == 0);     // 4
```

El siguiente ejemplo muestra la diferencia entre First y FirstOrDefault:

```
int firstBigError  = numbers.First          (n => n > 10);   // Exception
int firstBigNumber = numbers.FirstOrDefault (n => n > 10);   // 0
```

Para evitar una excepción, Single requiere exactamente un elemento que coincida; SingleOrDefault requiere uno o cero elementos que coincidan:

```
int onlyDivBy3 = numbers.Single (n => n % 3 == 0);   // 3
int divBy2Err  = numbers.Single (n => n % 2 == 0);   // Error: 2 & 4 match

int singleError = numbers.Single          (n => n > 10);      // Error
int noMatches   = numbers.SingleOrDefault (n => n > 10);      // 0
int divBy2Error = numbers.SingleOrDefault (n => n % 2 == 0);  // Error
```

Single es el más exigente en esta familia de operadores de elementos. FirstOrDefault y LastOrDefault son los más tolerantes.

En EF Core, Single se utiliza a menudo para recuperar una fila de una tabla mediante la clave primaria:

```
Customer cust = dataContext.Customers.Single (c => c.ID == 3);
```

ElementAt

Argument	Type
Source sequence	IEnumerable<TSource>
Index of element to return	int

ElementAt selecciona el enésimo elemento de la secuencia:

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int third      = numbers.ElementAt (2);           // 3
int tenthError = numbers.ElementAt (9);           // Exception
int tenth      = numbers.ElementAtOrDefault (9);  // 0
```

Enumerable.ElementAt está escrito de manera tal que si la secuencia de entrada implementa IList<T>, llama al indexador de IList<T>. De lo contrario, enumera n veces y luego devuelve el siguiente elemento. ElementAt no es compatible con EF Core.

MinBy y MaxBy

MinBy y MaxBy (introducidos en .NET 6) devuelven el elemento con el valor más pequeño o más grande, según lo determinado por un keySelector:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
Console.WriteLine (names.MaxBy (n => n.Length));   // Harry
```

En contraste, Min y Max (que se cubrirán en la siguiente sección) devuelven el valor más pequeño o más grande en sí mismo:

```
Console.WriteLine (names.Max   (n => n.Length));   // 5
```

Si dos o más elementos comparten un valor mínimo/máximo, MinBy/MaxBy devuelve el primero:

```
Console.WriteLine (names.MinBy (n => n.Length));   // Tom
```

Si la secuencia de entrada está vacía, MinBy y MaxBy devuelven null si el tipo de elemento es nullable (o lanzan una excepción si el tipo de elemento no es nullable).

DefaultIfEmpty

DefaultIfEmpty devuelve una secuencia que contiene un solo elemento cuyo valor es default(TSource) si la secuencia de entrada no tiene elementos; de lo contrario, devuelve la secuencia de entrada sin cambios. Se utiliza esto al escribir joins externos planos: ver "Joins externos con SelectMany" en la página 467 y "Joins externos planos" en la página 473.

Métodos de agregación

`IEnumerable<TSource>→scalar`

Method	Description	SQL equivalents
`Count`, `LongCount`	Returns the number of elements in the input sequence, optionally satisfying a predicate	`COUNT (...)`
`Min, Max`	Returns the smallest or largest element in the sequence	`MIN (...)`, `MAX (...)`
`Sum, Average`	Calculates a numeric sum or average over elements in the sequence	`SUM (...)`, `AVG (...)`
`Aggregate`	Performs a custom aggregation	Exception thrown

Count y LongCount

Argument	Type
Source sequence	`IEnumerable<TSource>`
Predicate (optional)	`TSource => bool`

Count simplemente enumera una secuencia, devolviendo el número de elementos:

```
int fullCount = new int[] { 5, 6, 7 }.Count();     // 3
```

La implementación interna de Enumerable.Count prueba la secuencia de entrada para ver si implementa ICollection<T>. Si lo hace, simplemente llama a ICollection<T>.Count; de lo contrario, enumera cada elemento, incrementando un contador.

Opcionalmente, puedes proporcionar un predicado:

```
int digitCount = "pa55w0rd".Count (c => char.IsDigit (c));   // 3
```

LongCount hace el mismo trabajo que Count, pero devuelve un entero de 64 bits, lo que permite secuencias con más de dos mil millones de elementos.

Min y Max

Argument	Type
Source sequence	IEnumerable<TSource>
Result selector (optional)	TSource => TResult

Min y Max devuelven el elemento más pequeño o más grande de una secuencia:

```
int[] numbers = { 28, 32, 14 };
int smallest = numbers.Min();  // 14;
int largest  = numbers.Max();  // 32;
```

Si incluyes una expresión de selector, cada elemento se proyecta primero:

```
int smallest = numbers.Max (n => n % 10);  // 8;
```

Una expresión de selector es obligatoria si los elementos en sí no son intrínsecamente comparables, es decir, si no implementan IComparable<T>:

```
Purchase runtimeError = dbContext.Purchases.Min ();             // Error
decimal? lowestPrice = dbContext.Purchases.Min (p => p.Price);  // OK
```

Una expresión de selector determina no solo cómo se comparan los elementos, sino también el resultado final. En el ejemplo anterior, el resultado final es un valor decimal, no un objeto de compra. Para obtener la compra más barata, necesitas una subconsulta:

```
Purchase cheapest = dbContext.Purchases
  .Where (p => p.Price == dbContext.Purchases.Min (p2 => p2.Price))
  .FirstOrDefault();
```

En este caso, también podrías formular la consulta sin una agregación utilizando un OrderBy seguido de FirstOrDefault.

Sum y Average

Argument	Type
Source sequence	IEnumerable<TSource>
Result selector (optional)	TSource => TResult

Sum y Average son operadores de agregación que se utilizan de manera similar a Min y Max:

```
decimal[] numbers  = { 3, 4, 8 };
decimal sumTotal   = numbers.Sum();              // 15
decimal average    = numbers.Average();          // 5   (mean value)
```

Lo siguiente devuelve la longitud total de cada una de las cadenas en el array de nombres:

```
int combinedLength = names.Sum (s => s.Length);   // 19
```

Suma y Promedio son bastante restrictivos en su tipificación. Sus definiciones están cableadas directamente a cada uno de los tipos numéricos (int, long, float, double, decimal y sus versiones anulables). En contraste, Min y Max pueden operar directamente en cualquier cosa que implemente IComparable<T>, como por ejemplo una cadena.

Además, Average siempre devuelve decimal, float o double, según la siguiente tabla:

Selector type	Result type
decimal	decimal
float	float
int, long, double	double

Esto significa que lo siguiente no compila ("no se puede convertir double a int"):

```
int avg = new int[] { 3, 4 }.Average();
```

Pero esto sí compilará:

```
double avg = new int[] { 3, 4 }.Average();   // 3.5
```

Average aumenta implícitamente los valores de entrada para evitar la pérdida de precisión. En este ejemplo, obtuvimos un promedio de enteros y obtuvimos 3.5 sin necesidad de recurrir a una conversión de elementos de entrada:

```
double avg = numbers.Average (n => (double) n);
```

Cuando se consulta una base de datos, Sum y Average se traducen a las agregaciones SQL estándar. La siguiente consulta devuelve los clientes cuya compra promedio fue superior a $500:

```
from c in dbContext.Customers
where c.Purchases.Average (p => p.Price) > 500
select c.Name;
```

Aggregate

Aggregate te permite especificar un algoritmo de acumulación personalizado para implementar agregaciones inusuales. Aggregate no es compatible con EF Core y es algo especializado en sus casos de uso. Lo siguiente demuestra cómo Aggregate puede realizar el trabajo de Sum:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate (0, (total, n) => total + n);   // 6
```

El primer argumento de Aggregate es la semilla, desde la cual comienza la acumulación. El segundo argumento es una expresión para actualizar el valor acumulado, dada un nuevo elemento. Opcionalmente, puedes proporcionar un tercer argumento para proyectar el valor final del resultado a partir del valor acumulado.



La mayoría de los problemas para los que se ha diseñado Aggregate se pueden resolver fácilmente con un bucle foreach y con una sintaxis más familiar. La ventaja de usar Aggregate es que, con agregaciones grandes o complejas, puedes paralelizar automáticamente la operación con PLINQ (consultar el Capítulo 22).

Agregaciones sin semilla (Unseeded aggregations)

Puedes omitir el valor de semilla al llamar a Aggregate, en cuyo caso el primer elemento se convierte en la semilla implícita y la agregación se realiza a partir del segundo elemento. Aquí tienes el ejemplo anterior sin semilla:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate ((total, n) => total + n);   // 6
```

Esto da el mismo resultado que antes, pero en realidad estamos realizando un cálculo diferente. Antes, estábamos calculando 0 + 1 + 2 + 3; ahora estamos calculando 1 + 2 + 3. Podemos ilustrar mejor la diferencia multiplicando en lugar de sumar:

```
int[] numbers = { 1, 2, 3 };
int x = numbers.Aggregate (0, (prod, n) => prod * n);   // 0*1*2*3 = 0
int y = numbers.Aggregate (   (prod, n) => prod * n);   //   1*2*3 = 6
```

Como verás en el Capítulo 22, las agregaciones sin semilla tienen la ventaja de poder paralelizarse sin necesidad de utilizar sobrecargas especiales. Sin embargo, hay algunas trampas con las agregaciones sin semilla.

Trampas con las agregaciones sin semilla

Los métodos de agregación sin semilla están diseñados para ser utilizados con delegados que sean conmutativos y asociativos. Si se utilizan de otra manera, el resultado será o bien poco intuitivo (en consultas normales) o no determinista (en el caso de que paralelices la consulta con PLINQ). Por ejemplo, considera la siguiente función:

```
(total, n) => total + n * n
```

Esta función no es ni conmutativa ni asociativa. (Por ejemplo, 1 + 2 * 2 != 2 + 1 * 1). Veamos qué sucede cuando la utilizamos para sumar el cuadrado de los números 2, 3 y 4:

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate ((total, n) => total + n * n);    // 27
```

En lugar de calcular

```
2*2 + 3*3 + 4*4    // 29
```

calcula:

```
 2 + 3*3 + 4*4      // 27
```

Podemos solucionar esto de varias maneras. Primero, podríamos incluir 0 como primer elemento:

```
int[] numbers = { 0, 2, 3, 4 };
```

No solo es esto poco elegante, sino que también dará resultados incorrectos si se paraleliza, porque PLINQ utiliza la asociatividad asumida de la función al seleccionar múltiples elementos como semillas. Para ilustrar esto, si denotamos nuestra función de agregación de la siguiente manera:

```
f(total, n) => total + n * n
```

LINQ to Objects calcularía esto:

```
f(f(f(0, 2),3),4)
```

mientras que PLINQ podría hacer esto:

```
f(f(0,2),f(3,4))
```

con el siguiente resultado:

```
First partition:   a = 0 + 2*2  (= 4)
Second partition:  b = 3 + 4*4  (= 19)
Final result:          a + b*b  (= 365)
OR EVEN:               b + a*a  (= 35)
```

Existen dos buenas soluciones. La primera es convertir esto en una agregación con semilla, utilizando 0 como semilla. La única complicación es que con PLINQ, necesitaríamos utilizar una sobrecarga especial para que la consulta no se ejecute secuencialmente (consultar "Optimizing PLINQ" en la página 919).

La segunda solución es reestructurar la consulta de manera que la función de agregación sea conmutativa y asociativa:

```
int sum = numbers.Select (n => n * n).Aggregate ((total, n) => total + n);
```



Por supuesto, en escenarios tan simples, se puede (y se debe) utilizar el operador Sum en lugar de Aggregate:

```
int sum = numbers.Sum (n => n * n);
```

En realidad, se puede llegar bastante lejos solo con Sum y Average. Por ejemplo, se puede utilizar Average para calcular la raíz cuadrada media:

```
Math.Sqrt (numbers.Average (n => n * n))
```

Incluso se puede calcular la desviación estándar:

```
double mean = numbers.Average();
double sdev = Math.Sqrt (numbers.Average (n =>
              {
                double dif = n - mean;
                return dif * dif;
              }));
```

Ambos enfoques son seguros, eficientes y completamente paralelizables. En el Capítulo 22, se presenta un ejemplo práctico de una agregación personalizada que no se puede reducir a Sum o Average.

Cuantificadores

`IEnumerable<TSource>→bool`

Method	Description	SQL equivalents
`Contains`	Returns `true` if the input sequence contains the given element	`WHERE ... IN (...)`
`Any`	Returns `true` if any elements satisfy the given predicate	`WHERE ... IN (...)`
`All`	Returns `true` if all elements satisfy the given predicate	`WHERE (...)`
`SequenceEqual`	Returns `true` if the second sequence has identical elements to the input sequence	

Contains y Any

El método Contains acepta un argumento de tipo TSource; Any acepta un predicado opcional.

Contains devuelve true si el elemento dado está presente:

```
bool hasAThree = new int[] { 2, 3, 4 }.Contains (3);       // true;
```

Any devuelve true si la expresión dada es verdadera para al menos un elemento. Podemos reescribir la consulta anterior con Any de la siguiente manera:

```
bool hasAThree = new int[] { 2, 3, 4 }.Any (n => n == 3);  // true;
```

Any puede hacer todo lo que Contains puede hacer, y más:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Any (n => n > 10);  // false;
```

Llamar a Any sin un predicado devuelve true si la secuencia tiene uno o más elementos. Aquí hay otra forma de escribir la consulta anterior:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Where (n => n > 10).Any();
```

Any es particularmente útil en subconsultas y se utiliza a menudo al consultar bases de datos. Por ejemplo:

```
from c in dbContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select c
```

All y SequenceEqual

All devuelve true si todos los elementos satisfacen un predicado. Lo siguiente devuelve clientes cuyas compras son inferiores a $100:

```
dbContext.Customers.Where (c => c.Purchases.All (p => p.Price < 100));
```

SequenceEqual compara dos secuencias. Para que devuelva true, cada secuencia debe tener elementos idénticos, en el mismo orden. Opcionalmente, puedes proporcionar un comparador de igualdad; el predeterminado es EqualityComparer<T>.Default.

Métodos de generación

```
void→IEnumerable<TResult>
```

Method	Description
Empty	Creates an empty sequence
Repeat	Creates a sequence of repeating elements
Range	Creates a sequence of integers

Empty, Repeat y Range son métodos estáticos (no extensiones) que generan secuencias locales simples.

Empty

Empty genera una secuencia vacía y solo requiere un argumento de tipo:

```
foreach (string s in Enumerable.Empty<string>())
  Console.Write (s);                              // <nothing>
```

En combinación con el operador ??, Empty hace lo contrario de DefaultIfEmpty. Por ejemplo, supongamos que tenemos una matriz irregular de enteros y queremos obtener todos los enteros

en una lista plana. La siguiente consulta SelectMany falla si alguna de las matrices internas es nula:

```
int[][] numbers =
{
  new int[] { 1, 2, 3 },
  new int[] { 4, 5, 6 },
  null                    // this null makes the query below fail.
};

IEnumerable<int> flat = numbers.SelectMany (innerArray => innerArray);
```

Empty en combinación con ?? soluciona el problema:

```
IEnumerable<int> flat = numbers
  .SelectMany (innerArray => innerArray ?? Enumerable.Empty <int>());

foreach (int i in flat)
  Console.Write (i + " ");     // 1 2 3 4 5 6
```

Range y Repeat

Range acepta un índice de inicio y una cantidad (ambos enteros):

```
foreach (int i in Enumerable.Range (5, 3))
  Console.Write (i + " ");                    // 5 6 7
```

Repeat acepta un elemento para repetir y el número de repeticiones:

```
foreach (bool x in Enumerable.Repeat (true, 3))
  Console.Write (x + " ");                   // True True True
```

10 LINQ to XML

.NET proporciona varias APIs para trabajar con datos XML. La opción principal para el procesamiento de documentos XML de propósito general es LINQ to XML. LINQ to XML incluye un modelo de objetos de documento XML (DOM) ligero y compatible con LINQ, junto con un conjunto de operadores de consulta complementarios.

En este capítulo, nos centramos exclusivamente en LINQ to XML. En el Capítulo 11, cubriremos el lector/escritor de XML de un solo sentido, y en el suplemento en línea, cubriremos los tipos para trabajar con esquemas y hojas de estilo. .NET también incluye el DOM basado en XmlDocument, que no cubrimos en este capítulo.

El DOM de LINQ to XML está diseñado de manera extremadamente eficiente y con un alto rendimiento. Incluso sin LINQ, el DOM de LINQ to XML es valioso como una fachada ligera sobre las clases XmlReader y XmlWriter de bajo nivel.

Todos los tipos de LINQ to XML están definidos en el espacio de nombres System.Xml.Linq.

Resumen arquitectónico

Esta sección comienza con una introducción muy breve al concepto de un DOM y luego explica la justificación detrás del DOM de LINQ to XML.

¿Qué es un DOM?

Considera el siguiente archivo XML:

```
<?xml version="1.0" encoding="utf-8"?>
<customer id="123" status="archived">
  <firstname>Joe</firstname>
  <lastname>Bloggs</lastname>
</customer>
```

Como ocurre con todos los archivos XML, comenzamos con una declaración y luego con un elemento raíz, cuyo nombre es "customer". El elemento "customer" tiene dos atributos, cada uno con un nombre (id y status) y un valor ("123" y "archived"). Dentro de "customer", hay dos elementos secundarios, "firstname" y "lastname", cada uno con un contenido de texto simple ("Joe" y "Bloggs").

Cada uno de estos constructos: declaración, elemento, atributo, valor y contenido de texto, puede representarse con una clase. Y si estas clases tienen propiedades de colección para almacenar contenido secundario, podemos armar un árbol de objetos para describir completamente un documento. Esto se llama Modelo de Objetos de Documento, o DOM.

El DOM de LINQ to XML

LINQ to XML consta de dos cosas:

- Un DOM XML, que llamamos X-DOM
- Un conjunto de aproximadamente 10 operadores de consulta complementarios

Como podrías esperar, el X-DOM consiste en tipos como XDocument, XElement y XAttribute. Curiosamente, los tipos X-DOM no están vinculados a LINQ: puedes cargar, instanciar, actualizar y guardar un X-DOM sin escribir nunca una consulta de LINQ. A su vez, podrías usar LINQ para consultar un DOM creado con los tipos antiguos compatibles con W3C. Sin embargo, esto sería frustrante y limitante. La característica distintiva del X-DOM es que es compatible con LINQ, lo que significa:

- Tiene métodos que generan secuencias IEnumerable útiles sobre las cuales puedes realizar consultas.
- Sus constructores están diseñados de manera que puedes construir un árbol X-DOM a través de una proyección de LINQ.

Visión general del X-DOM

La Figura 10-1 muestra los tipos principales del X-DOM. El más utilizado de estos tipos es XElement. XObject es la raíz de la jerarquía de herencia; XElement y XDocument son las raíces de la jerarquía de contenedores.

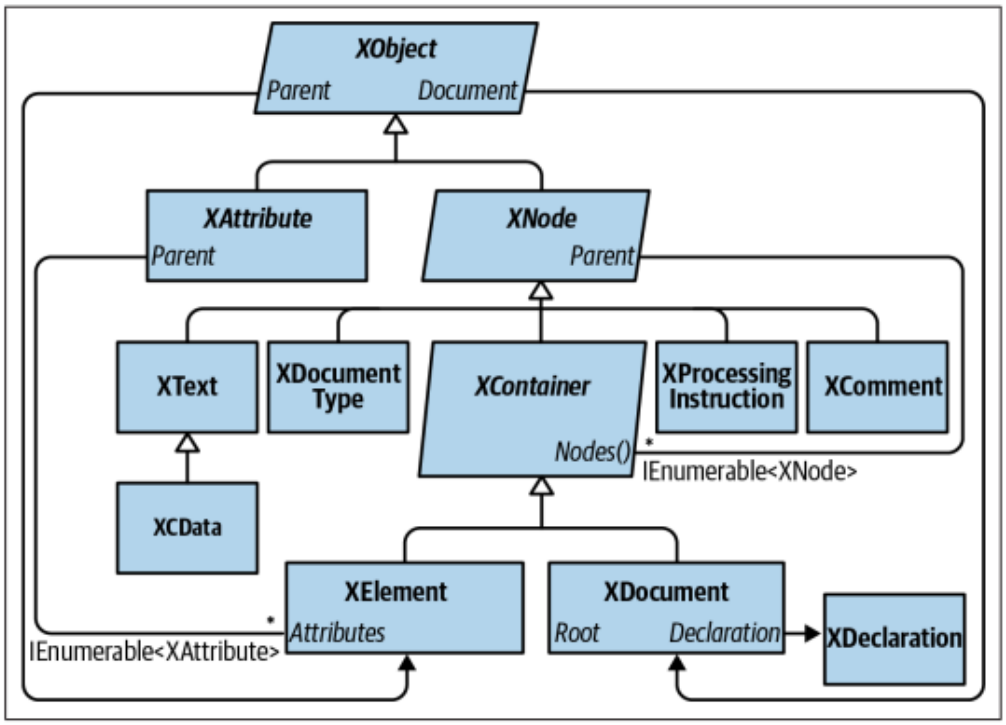


Figure 10-1. Core X-DOM types

La Figura 10-2 muestra el árbol X-DOM creado a partir del siguiente código:

```
string xml = @"<customer id='123' status='archived'>
                 <firstname>Joe</firstname>
                 <lastname>Bloggs<!--nice name--></lastname>
               </customer>";

XElement customer = XElement.Parse (xml);
```

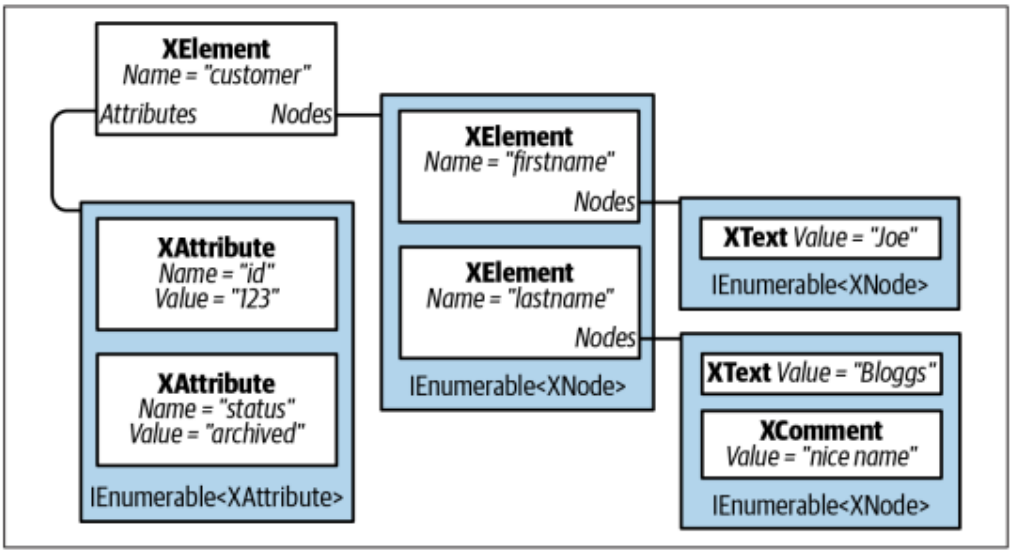


Figure 10-2. A simple X-DOM tree

XObject es la clase base abstracta para todo el contenido XML. Define un enlace al elemento padre en el árbol de contenedores, así como un XDocument opcional.

XNode es la clase base para la mayoría del contenido XML, excluyendo los atributos. La característica distintiva de XNode es que puede estar en una colección ordenada de XNodes de diferentes tipos. Por ejemplo, considera el siguiente XML:

```
<data>
  Hello world
  <subelement1/>
  <!--comment-->
  <subelement2/>
</data>
```

Dentro del elemento padre <data>, primero hay un nodo XText (Hello world), luego un nodo XElement, luego un nodo XComment y finalmente otro nodo XElement. En contraste, un XAttribute solo tolera otros XAttributes como pares.

Aunque un XNode puede acceder a su elemento padre XElement, no tiene concepto de nodos hijos, ya que esta es la tarea de su subclase XContainer. XContainer define miembros para tratar con los hijos y es la clase base abstracta para XElement y XDocument.

XElement introduce miembros para gestionar los atributos, así como un Nombre y un Valor. En el caso (bastante común) de que un elemento tenga un único nodo XText hijo, la propiedad Valor en XElement encapsula el contenido de este hijo tanto para operaciones de obtención como de asignación, evitando navegaciones innecesarias. Gracias a Valor, puedes evitar trabajar directamente con nodos XText en la mayoría de los casos.

XDocument representa la raíz de un árbol XML. Más precisamente, envuelve el XElement raíz, añadiendo una XDeclaration, instrucciones de procesamiento y otros detalles a nivel de raíz. A diferencia del DOM de la W3C, su uso es opcional: ¡puedes cargar, manipular y guardar un X-DOM sin necesidad de crear un XDocument! El hecho de no depender de XDocument también significa que puedes mover eficiente y fácilmente un subárbol de nodos a otra jerarquía X-DOM.

Carga y análisis

Tanto XElement como XDocument proporcionan métodos estáticos Load y Parse para construir un árbol X-DOM a partir de una fuente existente:

- Load construye un X-DOM a partir de un archivo, URI, flujo (Stream), lector de texto (TextReader) o lector XML (XmlReader).
- Parse construye un X-DOM a partir de una cadena de texto.

Por ejemplo:

```
XDocument fromWeb = XDocument.Load ("http://albahari.com/sample.xml");

XElement fromFile = XElement.Load (@"e:\media\somefile.xml");

XElement config = XElement.Parse (
@"<configuration>

   <client enabled='true'>
     <timeout>30</timeout>
   </client>
 </configuration>");
```

En las secciones posteriores, describiremos cómo recorrer y actualizar un X-DOM. Como vista previa rápida, aquí tienes cómo manipular el elemento de configuración que acabamos de poblar:

```
foreach (XElement child in config.Elements())
  Console.WriteLine (child.Name);                    // client

XElement client = config.Element ("client");

bool enabled = (bool) client.Attribute ("enabled");   // Read attribute
Console.WriteLine (enabled);                          // True
client.Attribute ("enabled").SetValue (!enabled);     // Update attribute

int timeout = (int) client.Element ("timeout");       // Read element
Console.WriteLine (timeout);                          // 30
client.Element ("timeout").SetValue (timeout * 2);    // Update element

client.Add (new XElement ("retries", 3));             // Add new element

Console.WriteLine (config);         // Implicitly call config.ToString()
```

Aquí está el resultado de ese último Console.WriteLine:

```
<configuration>
  <client enabled="false">
    <timeout>60</timeout>
    <retries>3</retries>
  </client>
</configuration>
```



XNode también proporciona un método estático ReadFrom que instancia y popula cualquier tipo de nodo a partir de un XmlReader. A diferencia de Load, se detiene después de

leer un nodo completo, por lo que puedes continuar leyendo manualmente desde el XmlReader posteriormente.

También puedes hacer lo contrario y usar un XmlReader o XmlWriter para leer o escribir un XNode, a través de sus métodos CreateReader y CreateWriter.

Describimos los lectores y escritores de XML y cómo usarlos con el X-DOM en el Capítulo 11.

Guardado y serialización

Llamar a ToString en cualquier nodo convierte su contenido en una cadena XML, con formato que incluye saltos de línea e indentación como acabamos de ver. (Puedes desactivar los saltos de línea y la indentación especificando SaveOptions.DisableFormatting al llamar a ToString).

XElement y XDocument también proporcionan un método Save que escribe un X-DOM en un archivo, Stream, TextWriter o XmlWriter. Si especificas un archivo, se escribirá automáticamente una declaración XML en el X-DOM. También hay un método WriteTo definido en la clase XNode, que acepta solo un XmlWriter.

Describimos con más detalle el manejo de las declaraciones XML al guardar en "Documentos y declaraciones" en la página 515.

Instanciar un X-DOM

En lugar de usar los métodos Load o Parse, puedes construir un árbol X-DOM instanciando manualmente objetos y agregándolos a un padre a través del método Add de XContainer.

Para construir un XElement y XAttribute, simplemente proporciona un nombre y un valor:

```
XElement lastName = new XElement ("lastname", "Bloggs");
lastName.Add (new XComment ("nice name"));

XElement customer = new XElement ("customer");
customer.Add (new XAttribute ("id", 123));
customer.Add (new XElement ("firstname", "Joe"));
customer.Add (lastName);

Console.WriteLine (customer.ToString());
```

Aquí está el resultado:

```
<customer id="123">
  <firstname>Joe</firstname>
  <lastname>Bloggs<!--nice name--></lastname>
</customer>
```

Un valor es opcional al construir un XElement, puedes proporcionar solo el nombre del elemento y agregar contenido más adelante. Observa que cuando proporcionamos un valor, una cadena simple fue suficiente, no necesitamos crear y agregar explícitamente un nodo hijo XText. El X-DOM hace este trabajo automáticamente, por lo que puedes lidiar simplemente con "valores".

Construcción funcional

En nuestro ejemplo anterior, es difícil deducir la estructura XML a partir del código. El X-DOM admite otro modo de instanciación, llamado construcción funcional (proveniente de la programación funcional). Con la construcción funcional, puedes construir un árbol completo en una sola expresión:

```
XElement customer =
  new XElement ("customer", new XAttribute ("id", 123),
    new XElement ("firstname", "joe"),
    new XElement ("lastname", "bloggs",
      new XComment ("nice name")
    )
  );
```

Esto tiene dos beneficios. Primero, el código se asemeja a la estructura del XML. Segundo, puede incorporarse en la cláusula select de una consulta LINQ. Por ejemplo, la siguiente consulta proyecta desde una clase de entidad de EF Core hacia un X-DOM:

```
XElement query =
  new XElement ("customers",
    from c in dbContext.Customers.AsEnumerable()
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("firstname", c.FirstName),
        new XElement ("lastname", c.LastName,
          new XComment ("nice name")
        )
      )
  );
```

Examinaremos esto con más detalle más adelante en este capítulo en "Proyectar en un X-DOM" en la página 525.

Especificar contenido

La construcción funcional es posible porque los constructores de XElement (y XDocument) están sobrecargados para aceptar un arreglo de objetos params:

```
public XElement (XName name, params object[] content)
```

Lo mismo es válido para el método Add en XContainer:

```
public void Add (params object[] content)
```

Por lo tanto, puedes especificar cualquier número de objetos hijos de cualquier tipo al construir o agregar a un X-DOM. Esto funciona porque cualquier cosa se considera contenido válido. Para ver cómo funciona, debemos examinar cómo se procesa internamente cada objeto de contenido. Estas son las decisiones tomadas por XContainer, en orden:

1. Si el objeto es nulo, se ignora.

2. Si el objeto se basa en XNode o XStreamingElement, se agrega tal cual a la colección de Nodes.

3. Si el objeto es un XAttribute, se agrega a la colección de Attributes.

4. Si el objeto es una cadena, se envuelve en un nodo XText y se agrega a Nodes.1

5. Si el objeto implementa IEnumerable, se enumera y se aplican las mismas reglas a cada elemento.

6. De lo contrario, el objeto se convierte a una cadena, se envuelve en un nodo XText y luego se agrega a Nodes.

Todo termina en uno de dos grupos: Nodes o Attributes. Además, cualquier objeto es un contenido válido porque siempre se puede llamar a ToString en él y tratarlo como un nodo XText.



Antes de llamar a ToString en un tipo arbitrario, XContainer primero verifica si es uno de los siguientes tipos:

```
float, double, decimal, bool,
DateTime, DateTimeOffset, TimeSpan
```

Si es así, llama a un método ToString adecuado con tipo en la clase de ayuda XmlConvert en lugar de llamar a ToString en el objeto mismo. Esto asegura que los datos sean redondeables y cumplan con las reglas estándar de formato XML.

Clonación automática profunda

Cuando se agrega un nodo o atributo a un elemento (ya sea mediante construcción funcional o mediante el método Add), la propiedad Parent del nodo o atributo se establece en ese elemento. Un nodo solo puede tener un elemento padre: si agregas un nodo que ya tiene un padre a un segundo padre, el nodo se clona automáticamente en profundidad. En el siguiente ejemplo, cada cliente tiene una copia separada de la dirección:

```
var address = new XElement ("address",
                  new XElement ("street", "Lawley St"),
                  new XElement ("town", "North Beach")
              );
var customer1 = new XElement ("customer1", address);
var customer2 = new XElement ("customer2", address);

customer1.Element ("address").Element ("street").Value = "Another St";
Console.WriteLine (
  customer2.Element ("address").Element ("street").Value);   // Lawley St
```

Esta duplicación automática mantiene la instantánea de objetos X-DOM libre de efectos secundarios, otro sello distintivo de la programación funcional.

Navegación y consulta

Como era de esperar, las clases XNode y XContainer definen métodos y propiedades para recorrer el árbol X-DOM. Sin embargo, a diferencia de un DOM convencional, estas funciones no devuelven una colección que implemente IList<T>. En su lugar, devuelven un solo valor o una secuencia que implementa IEnumerable<T>, sobre la cual se espera que ejecutes una consulta

LINQ (o la enumeres con un foreach). Esto permite realizar consultas avanzadas, así como tareas de navegación simples, utilizando la familiar sintaxis de consulta LINQ.


Los nombres de elementos y atributos distinguen entre mayúsculas y minúsculas en el X-DOM, al igual que en XML.

Navegación de nodos hijos

Return type	Members	Works on
XNode	FirstNode { get; }	XContainer
	LastNode { get; }	XContainer
IEnumerable<XNode>	Nodes()	XContainer*
	DescendantNodes()	XContainer*
	DescendantNodesAndSelf()	XElement*
XElement	Element (XName)	XContainer
IEnumerable<XElement>	Elements()	XContainer*
	Elements (XName)	XContainer*
	Descendants()	XContainer*
	Descendants (XName)	XContainer*
	DescendantsAndSelf()	XElement*
	DescendantsAndSelf (XName)	XElement*
bool	HasElements { get; }	XElement


Las funciones marcadas con un asterisco en la tercera columna de esta y otras tablas también operan en secuencias del mismo tipo. Por ejemplo, puedes llamar a Nodes en un XContainer o en una secuencia de objetos XContainer. Esto es posible gracias a los métodos de extensión definidos en System.Xml.Linq, los operadores de consulta suplementarios de los que hablamos en la descripción general.

FirstNode, LastNode y Nodes

FirstNode y LastNode te brindan acceso directo al primer o último nodo hijo; Nodes devuelve todos los hijos como una secuencia. Las tres funciones solo consideran los descendientes directos:

```
var bench = new XElement ("bench",
              new XElement ("toolbox",
                new XElement ("handtool", "Hammer"),
                new XElement ("handtool", "Rasp")
              ),
              new XElement ("toolbox",
                new XElement ("handtool", "Saw"),
                new XElement ("powertool", "Nailgun")
              ),
              new XComment ("Be careful with the nailgun")
            );

foreach (XNode node in bench.Nodes())
  Console.WriteLine (node.ToString (SaveOptions.DisableFormatting) + ".");
```

Recuperando elementos (Retrieving elements)

El método Elements devuelve solo los nodos hijos del tipo XElement:

```
foreach (XElement e in bench.Elements())
  Console.WriteLine (e.Name + "=" + e.Value);     // toolbox=HammerRasp
                                                  // toolbox=SawNailgun
```

La siguiente consulta LINQ encuentra el "toolbox" con la herramienta de clavos:

```
IEnumerable<string> query =
  from toolbox in bench.Elements()
  where toolbox.Elements().Any (tool => tool.Value == "Nailgun")
  select toolbox.Value;

RESULT: { "SawNailgun" }
```

El siguiente ejemplo utiliza una consulta SelectMany para recuperar las herramientas manuales de todas las cajas de herramientas:

```
IEnumerable<string> query =
  from toolbox in bench.Elements()
  from tool in toolbox.Elements()
  where tool.Name == "handtool"
  select tool.Value;

RESULT: { "Hammer", "Rasp", "Saw" }
```



Elements en sí mismo es equivalente a una consulta LINQ en Nodes. Nuestra consulta anterior podría comenzar de la siguiente manera:

```
from toolbox in bench.Nodes().OfType<XElement>()
where ...
```

Elements también puede devolver solo los elementos con un nombre dado:

```
int x = bench.Elements ("toolbox").Count();     // 2
```

Esto es equivalente a lo siguiente:

```
int x = bench.Elements().Where (e => e.Name == "toolbox").Count();  // 2
```

Elements también se define como un método de extensión que acepta IEnumerable<XContainer> o, más precisamente, acepta un argumento de este tipo:

```
IEnumerable<T> where T : XContainer
```

Esto permite que funcione con secuencias de elementos también. Usando este método, podemos reescribir la consulta que encuentra las herramientas manuales en todas las cajas de herramientas de la siguiente manera:

```
from tool in bench.Elements ("toolbox").Elements ("handtool")
select tool.Value;
```

La primera llamada a Elements se vincula al método de instancia de XContainer; la segunda llamada a Elements se vincula al método de extensión.

Obteniendo un solo elemento

El método Element (singular) devuelve el primer elemento coincidente con el nombre proporcionado. Element es útil para una navegación sencilla, como se muestra a continuación:

```
XElement settings = XElement.Load ("databaseSettings.xml");
string cx = settings.Element ("database").Element ("connectString").Value;
```

Element es equivalente a llamar a Elements() y luego aplicar el operador de consulta FirstOrDefault de LINQ con un predicado que coincida con el nombre. Element devuelve null si el elemento solicitado no existe.



Element("xyz").Value lanzará una NullReferenceException si el elemento xyz no existe. Si prefieres obtener null en lugar de una excepción, puedes utilizar el operador de condicional nulo: Element("xyz")?.Value. También puedes convertir el XElement a una cadena en lugar de consultar su propiedad Value. En otras palabras:

```
string xyz = (string) settings.Element ("xyz");
```

Esto funciona porque XElement define una conversión explícita a cadena, específicamente para este propósito.

Obteniendo descendientes

XContainer también proporciona los métodos Descendants y DescendantNodes que devuelven los elementos o nodos secundarios y todos sus descendientes, y así sucesivamente (todo el

árbol). Descendants acepta un nombre de elemento opcional. Volviendo a nuestro ejemplo anterior, podemos usar Descendants para encontrar todas las herramientas de mano:

```
Console.WriteLine (bench.Descendants ("handtool").Count());  // 3
```

Tanto los nodos padres como los nodos hoja se incluyen, como se muestra en el siguiente ejemplo:

```
foreach (XNode node in bench.DescendantNodes())
  Console.WriteLine (node.ToString (SaveOptions.DisableFormatting));
```

Aquí está la salida:

```
<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>
<handtool>Hammer</handtool>
Hammer
<handtool>Rasp</handtool>
Rasp
<toolbox><handtool>Saw</handtool><powertool>Nailgun</powertool></toolbox>
<handtool>Saw</handtool>
Saw
<powertool>Nailgun</powertool>
Nailgun
<!--Be careful with the nailgun-->
```

La siguiente consulta extrae todos los comentarios en cualquier parte del X-DOM que contienen la palabra "cuidado":

```
IEnumerable<string> query =
  from c in bench.DescendantNodes().OfType<XComment>()
  where c.Value.Contains ("careful")
  orderby c.Value
  select c.Value;
```

Navegación al elemento padre

Todos los XNodes tienen una propiedad Parent y métodos AncestorXXX para la navegación al elemento padre. El elemento padre siempre es un XElement:

Return type	Members	Works on
XElement	Parent { get; }	XNode
Enumerable<XElement>	Ancestors()	XNode
	Ancestors (XName)	XNode
	AncestorsAndSelf()	XElement
	AncestorsAndSelf (XName)	XElement

Si x es un XElement, lo siguiente siempre imprime true:

```
foreach (XNode child in x.Nodes())
  Console.WriteLine (child.Parent == x);
```

Sin embargo, esto no es cierto si x es un XDocument. XDocument es peculiar: puede tener hijos, pero nunca puede ser el padre de nadie. Para acceder al XDocument, en su lugar se utiliza la propiedad Document; esto funciona en cualquier objeto del árbol X-DOM.

Ancestors devuelve una secuencia cuyo primer elemento es el elemento padre (Parent) y cuyo siguiente elemento es el elemento padre del padre (Parent.Parent), y así sucesivamente hasta llegar al elemento raíz.

 Puedes navegar hasta el elemento raíz con la consulta LINQ AncestorsAndSelf().Last().

Otra forma de lograr lo mismo es llamar a Document.Root, aunque esto solo funciona si hay un XDocument presente.

Navegación entre nodos pares

Return type	Members	Defined in
bool	IsBefore (XNode node)	XNode
	IsAfter (XNode node)	XNode
XNode	PreviousNode { get; }	XNode
	NextNode { get; }	XNode
IEnumerable<XNode>	NodesBeforeSelf()	XNode
	NodesAfterSelf()	XNode
IEnumerable<XElement>	ElementsBeforeSelf()	XNode
	ElementsBeforeSelf (XName name)	XNode
	ElementsAfterSelf()	XNode
	ElementsAfterSelf (XName name)	XNode

Con PreviousNode y NextNode (y FirstNode/LastNode), puedes recorrer los nodos con la sensación de una lista enlazada. Esto no es una coincidencia: internamente, los nodos se almacenan en una lista enlazada.

 XNode utiliza internamente una lista enlazada simple, por lo que PreviousNode no es eficiente.

Navegación de atributos

Return type	Members	Defined in
bool	HasAttributes { get; }	XElement
XAttribute	Attribute (XName name)	XElement
	FirstAttribute { get; }	XElement
	LastAttribute { get; }	XElement
IEnumerable<XAttribute>	Attributes()	XElement
	Attributes (XName name)	XElement

Además, XAttribute define las propiedades PreviousAttribute y NextAttribute, así como Parent.

El método Attributes que acepta un nombre devuelve una secuencia con cero o un elemento; un elemento no puede tener nombres de atributos duplicados en XML.

Actualización de un X-DOM

Puedes actualizar elementos y atributos de las siguientes maneras:

- Llamar a SetValue o reasignar la propiedad Value.
- Llamar a SetElementValue o SetAttributeValue.
- Llamar a uno de los métodos RemoveXXX.
- Llamar a uno de los métodos AddXXX o ReplaceXXX, especificando un nuevo contenido.

También puedes reasignar la propiedad Name en objetos XElement.

Actualizaciones de Valor Simple

Members	Works on
SetValue (object value)	XElement, XAttribute
Value { get; set }	XElement, XAttribute

El método SetValue reemplaza el contenido de un elemento o atributo con un valor simple.

La propiedad Value hace lo mismo pero solo acepta datos de tipo string. Describiremos ambas funciones en detalle más adelante en "Trabajando con Valores" en la página 513. Un efecto de llamar a SetValue (o reasignar Value) es que reemplaza a todos los nodos hijos:

```
XElement settings = new XElement ("settings",
                      new XElement ("timeout", 30)
                    );
settings.SetValue ("blah");
Console.WriteLine (settings.ToString());  // <settings>blah</settings>
```

Actualización de Nodos Hijos y Atributos

Category	Members	Works on
Add	Add (params object[] content)	XContainer
	AddFirst (params object[] content)	XContainer
Remove	RemoveNodes()	XContainer
	RemoveAttributes()	XElement
	RemoveAll()	XElement
Update	ReplaceNodes (params object[] content)	XContainer
	ReplaceAttributes (params object[] content)	XElement
	ReplaceAll (params object[] content	XElement
	SetElementValue (XName name, object value)	XElement
	SetAttributeValue (XName name, object value)	XElement

Los métodos más convenientes en este grupo son los dos últimos: SetElementValue y SetAttributeValue. Sirven como atajos para instanciar un XElement o XAttribute y luego agregarlo a un elemento padre, reemplazando cualquier elemento o atributo existente con ese nombre:

```
XElement settings = new XElement ("settings");
settings.SetElementValue ("timeout", 30);     // Adds child node
settings.SetElementValue ("timeout", 60);     // Update it to 60
```

Add agrega un nodo hijo a un elemento o documento. AddFirst hace lo mismo pero inserta al principio de la colección en lugar de al final.

Puedes eliminar todos los nodos hijos o atributos de golpe con RemoveNodes o RemoveAttributes. RemoveAll es equivalente a llamar a ambos métodos.

Los métodos ReplaceXXX son equivalentes a la remoción y luego adición. Toman una instantánea del contenido de entrada, por lo que e.ReplaceNodes(e.Nodes()) funciona como se espera.

Actualización a través del Elemento Padre

Members	Works on
AddBeforeSelf (params object[] content)	XNode
AddAfterSelf (params object[] content)	XNode
Remove()	XNode, XAttribute
ReplaceWith (params object[] content)	XNode

Los métodos AddBeforeSelf, AddAfterSelf, Remove y ReplaceWith no operan en los hijos del nodo. En su lugar, operan en la colección a la que pertenece el nodo. Esto requiere que el nodo tenga un elemento padre, de lo contrario se lanzará una excepción. AddBeforeSelf y AddAfterSelf son útiles para insertar un nodo en una posición arbitraria:

```
XElement items = new XElement ("items",
                   new XElement ("one"),
                   new XElement ("three")
                 );
items.FirstNode.AddAfterSelf (new XElement ("two"));
```

Aquí está el resultado:

```
<items><one /><two /><three /></items>
```

Insertar en una posición arbitraria dentro de una larga secuencia de elementos es eficiente porque los nodos se almacenan internamente en una lista enlazada.

El método Remove elimina el nodo actual de su elemento padre. ReplaceWith hace lo mismo y luego inserta otro contenido en la misma posición:

```
XElement items = XElement.Parse ("<items><one/><two/><three/></items>");
items.FirstNode.ReplaceWith (new XComment ("One was here"));
```

Aquí está el resultado:

```
<items><!--one was here--><two /><three /></items>
```

Eliminar una secuencia de nodos o atributos

Gracias a los métodos de extensión en System.Xml.Linq, también puedes llamar a Remove en una secuencia de nodos o atributos. Considera este X-DOM:

```
XElement contacts = XElement.Parse (
@"<contacts>
    <customer name='Mary'/>
    <customer name='Chris' archived='true'/>
    <supplier name='Susan'>
      <phone archived='true'>012345678<!--confidential--></phone>
    </supplier>
  </contacts>");
```

Lo siguiente elimina todos los clientes:

```
contacts.Elements ("customer").Remove();
```

Lo siguiente elimina todos los contactos archivados (entonces Chris desaparece):

```
contacts.Elements().Where (e => (bool?) e.Attribute ("archived") == true)
                   .Remove();
```

Si reemplazamos Elements() con Descendants(), todos los elementos archivados en todo el DOM desaparecerían, dando como resultado esto:

```
<contacts>
  <customer name="Mary" />
  <supplier name="Susan" />
</contacts>
```

El siguiente ejemplo elimina todos los contactos que contienen el comentario "confidential" en cualquier parte de su árbol:

```
contacts.Elements().Where (e => e.DescendantNodes()
                                 .OfType<XComment>()
                                 .Any (c => c.Value == "confidential")
                           ).Remove();
```

Este es el resultado:

```
<contacts>
  <customer name="Mary" />
  <customer name="Chris" archived="true" />
</contacts>
```

Contrastando con la siguiente consulta más simple, que elimina todos los nodos de comentario del árbol:

```
contacts.DescendantNodes().OfType<XComment>().Remove();
```



Internamente, el método Remove primero lee todos los elementos coincidentes en una lista temporal y luego itera sobre esa lista temporal para realizar las eliminaciones. Esto evita errores que podrían ocurrir al eliminar y consultar al mismo tiempo.

Trabajando con valores

Tanto XElement como XAttribute tienen una propiedad Value de tipo string. Si un elemento tiene un solo nodo secundario XText, la propiedad Value de XElement actúa como un acceso conveniente al contenido de ese nodo. Con XAttribute, la propiedad Value es simplemente el valor del atributo.

A pesar de las diferencias de almacenamiento, el X-DOM proporciona un conjunto consistente de operaciones para trabajar con valores de elementos y atributos.

Asignando valores

Hay dos formas de asignar un valor: llamar a SetValue o asignar la propiedad Value. SetValue es más flexible porque acepta no solo cadenas, sino también otros tipos de datos simples:

```
var e = new XElement ("date", DateTime.Now);
e.SetValue (DateTime.Now.AddDays(1));
Console.Write (e.Value);              // 2019-10-02T16:39:10.734375+09:00
```

En lugar de eso, podríamos haber establecido directamente la propiedad Value del elemento, pero eso significaría convertir manualmente DateTime a una cadena. Esto es más complicado que llamar a ToString, ya que requiere el uso de XmlConvert para obtener un resultado compatible con XML.

Cuando se pasa un valor al constructor de XElement o XAttribute, también se realiza la conversión automática para tipos que no son de cadena. Esto asegura que los DateTimes se

formateen correctamente; true se escribe en minúsculas y double.NegativeInfinity se escribe como "-INF".

Obteniendo valores

Para convertir un valor a un tipo base, simplemente se realiza una conversión directa de XElement o XAttribute al tipo deseado. Parece que no debería funcionar, ¡pero lo hace! Por ejemplo:

```
XElement e = new XElement ("now", DateTime.Now);
DateTime dt = (DateTime) e;

XAttribute a = new XAttribute ("resolution", 1.234);
double res = (double) a;
```

Un elemento o atributo no almacena DateTimes o números de forma nativa, siempre se almacenan como texto y luego se analizan según sea necesario. Además, no "recuerdan" el tipo original, por lo que debes realizar la conversión correctamente para evitar errores en tiempo de ejecución. Para hacer que tu código sea robusto al trabajar con valores, puedes colocar la conversión en un bloque try/catch, capturando una excepción de tipo FormatException.

Las conversiones explícitas en XElement y XAttribute pueden analizar los siguientes tipos:

- Todos los tipos numéricos estándar.
- string, bool, DateTime, DateTimeOffset, TimeSpan y Guid.
- Versiones Nullable<> de los tipos de valor mencionados anteriormente.

La conversión a un tipo nullable es útil en conjunto con los métodos Element y Attribute, porque si el nombre solicitado no existe, la conversión aún funciona. Por ejemplo, si x no tiene un elemento "timeout", la primera línea genera un error en tiempo de ejecución y la segunda línea no lo hace:

```
int timeout = (int) x.Element ("timeout");      // Error
int? timeout = (int?) x.Element ("timeout");    // OK; timeout is null.
```

Puedes eliminar el tipo nullable en el resultado final utilizando el operador ?? (operador de fusión nula). Lo siguiente se evalúa como 1.0 si el atributo "resolution" no existe:

```
double resolution = (double?) x.Attribute ("resolution") ?? 1.0;
```

Sin embargo, la conversión a un tipo nullable no te evitará problemas si el elemento o atributo existe pero tiene un valor vacío (o con formato incorrecto). Para esto, debes capturar una excepción de tipo FormatException.

También puedes utilizar conversiones en consultas LINQ. Lo siguiente devuelve "John":

```
var data = XElement.Parse (
  @"<data>
      <customer id='1' name='Mary' credit='100' />
      <customer id='2' name='John' credit='150' />
      <customer id='3' name='Anne' />
    </data>");

IEnumerable<string> query = from cust in data.Elements()
                            where (int?) cust.Attribute ("credit") > 100
                            select cust.Attribute ("name").Value;
```

La conversión a int nullable evita una NullReferenceException en el caso de Anne, quien no tiene un atributo de crédito. Otra solución sería agregar un predicado a la cláusula where:

```
where cust.Attributes ("credit").Any() && (int) cust.Attribute...
```

Los mismos principios se aplican al consultar valores de elementos.

Valores y nodos de contenido mixto

Dado el valor de Value, es posible que te preguntes cuándo necesitarías lidiar directamente con nodos XText. La respuesta es cuando tienes contenido mixto. Por ejemplo:

Una propiedad de valor simple no es suficiente para capturar el contenido de un elemento "summary". El elemento "summary" contiene tres hijos: un nodo XText, seguido de un XElement y luego otro nodo XText. Aquí tienes cómo construirlo:

```
XElement summary = new XElement ("summary",
                     new XText ("An XAttribute is "),
                     new XElement ("bold", "not"),
                     new XText (" an XNode")
                   );
```

Curiosamente, aún podemos consultar el valor de "summary" sin obtener una excepción. En su lugar, obtenemos una concatenación del valor de cada hijo:

```
An XAttribute is not an XNode
```

También es posible reasignar el valor de "summary", pero esto implica reemplazar todos los hijos anteriores con un nuevo nodo XText único.

Concatenación automática de XText

Cuando agregas contenido simple a un XElement, el X-DOM lo agrega al nodo XText existente en lugar de crear uno nuevo. En los siguientes ejemplos, "e1" y "e2" terminan teniendo un solo hijo XText cuyo valor es "HelloWorld":

```
var e1 = new XElement ("test", "Hello"); e1.Add ("World");
var e2 = new XElement ("test", "Hello", "World");
```

Sin embargo, si creas específicamente nodos XText, terminarás con múltiples hijos:

```
var e = new XElement ("test", new XText ("Hello"), new XText ("World"));
Console.WriteLine (e.Value);             // HelloWorld
Console.WriteLine (e.Nodes().Count());   // 2
```

XElement no concatena los dos nodos XText, por lo que se preservan las identidades de los objetos nodos.

Documentos y Declaraciones

XDocument

Como mencionamos anteriormente, un XDocument envuelve un XElement raíz y te permite agregar una XDeclaration, instrucciones de procesamiento, un tipo de documento y comentarios de nivel raíz. Un XDocument es opcional y se puede ignorar u omitir: a diferencia del DOM de W3C, no sirve como pegamento para mantener todo junto.

Un XDocument proporciona los mismos constructores funcionales que XElement. Y como se basa en XContainer, también admite los métodos AddXXX, RemoveXXX y ReplaceXXX. Sin embargo, a diferencia de XElement, un XDocument solo puede aceptar un contenido limitado:

- Un solo objeto XElement (la "raíz").
- Un solo objeto XDeclaration.
- Un solo objeto XDocumentType (para hacer referencia a una definición de tipo de documento [DTD]).
- Cualquier número de objetos XProcessingInstruction.
- Cualquier número de objetos XComment.



De estos, solo el XElement raíz es obligatorio para tener un XDocument válido. La XDeclaration es opcional; si se omite, se aplicarán configuraciones predeterminadas durante la serialización.

El XDocument más simple válido tiene solo un elemento raíz:

```
var doc = new XDocument (
            new XElement ("test", "data")
          );
```

Observa que no incluimos un objeto XDeclaration. Sin embargo, el archivo generado al llamar a `doc.Save` aún contendrá una declaración XML, ya que se genera una por defecto.

El siguiente ejemplo produce un archivo XHTML simple pero correcto, ilustrando todos los elementos que un XDocument puede aceptar:

```
var styleInstruction = new XProcessingInstruction (
  "xml-stylesheet", "href='styles.css' type='text/css'");

var docType = new XDocumentType ("html",
  "-//W3C//DTD XHTML 1.0 Strict//EN",
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd", null);

XNamespace ns = "http://www.w3.org/1999/xhtml";
var root =
  new XElement (ns + "html",
    new XElement (ns + "head",
      new XElement (ns + "title", "An XHTML page")),
    new XElement (ns + "body",
      new XElement (ns + "p", "This is the content"))
  );

var doc =
  new XDocument (
    new XDeclaration ("1.0", "utf-8", "no"),
    new XComment ("Reference a stylesheet"),
    styleInstruction,
    docType,
    root);

doc.Save ("test.html");
```

El archivo test.html resultante se verá así:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!--Reference a stylesheet-->
<?xml-stylesheet href='styles.css' type='text/css'?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>An XHTML page</title>
  </head>
  <body>
    <p>This is the content</p>
  </body>
</html>
```

XDocument tiene una propiedad llamada "Root" que sirve como acceso directo al único XElement del documento. El enlace inverso se proporciona a través de la propiedad "Document" de XObject, que funciona para todos los objetos en el árbol:

```
Console.WriteLine (doc.Root.Name.LocalName);          // html
XElement bodyNode = doc.Root.Element (ns + "body");
Console.WriteLine (bodyNode.Document == doc);         // True
```

Recuerda que los hijos de un documento no tienen un "Parent":

```
Console.WriteLine (doc.Root.Parent == null);          // True
foreach (XNode node in doc.Nodes())
  Console.Write (node.Parent == null);                // TrueTrueTrueTrue
```



Una XDeclaration no es un XNode y no aparece en la colección "Nodes" del documento, a diferencia de los comentarios, las instrucciones de procesamiento y el elemento raíz. En su lugar, se asigna a una propiedad específica llamada "Declaration". Por eso, en el último ejemplo, "True" se repite cuatro veces y no cinco.

Declaraciones XML

Un archivo XML estándar comienza con una declaración como la siguiente:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
```

Una declaración XML garantiza que el archivo se analice y se comprenda correctamente por un lector. XElement y XDocument siguen estas reglas al emitir declaraciones XML:

- Llamar a "Save" con un nombre de archivo siempre escribe una declaración.
- Llamar a "Save" con un XmlWriter escribe una declaración a menos que se indique lo contrario en el XmlWriter.
- El método ToString nunca emite una declaración XML.



Puedes indicarle a un XmlWriter que no produzca una declaración configurando las propiedades "OmitXmlDeclaration" y "ConformanceLevel" de un objeto XmlWriterSettings al construir el XmlWriter. Esto se describe en el Capítulo 11.

La presencia o ausencia de un objeto XDeclaration no tiene efecto en si se escribe o no una declaración XML. El propósito de un XDeclaration es dar pistas a la serialización XML en dos aspectos:

- Qué codificación de texto utilizar.
- Qué poner en los atributos de codificación y standalone de la declaración XML (si se debe escribir una declaración).

El constructor de XDeclaration acepta tres argumentos, que corresponden a los atributos version, encoding y standalone. En el siguiente ejemplo, el archivo test.xml se codifica en UTF-16:

```
var doc = new XDocument (
            new XDeclaration ("1.0", "utf-16", "yes"),
            new XElement ("test", "data")
          );
doc.Save ("test.xml");
```



Sin embargo, lo que especifiques para la versión XML es ignorado por el escritor XML, ya que siempre escribe "1.0".

La codificación debe utilizar un código IETF como "utf-16", tal como aparecería en la declaración XML.

Escritura de una declaración en una cadena de texto

Supongamos que queremos serializar un XDocument a una cadena de texto, incluyendo la declaración XML. Dado que ToString no escribe una declaración, necesitaríamos utilizar un XmlWriter en su lugar:

```
var doc = new XDocument (
            new XDeclaration ("1.0", "utf-8", "yes"),
            new XElement ("test", "data")
          );
var output = new StringBuilder();
var settings = new XmlWriterSettings { Indent = true };
using (XmlWriter xw = XmlWriter.Create (output, settings))
  doc.Save (xw);
Console.WriteLine (output.ToString());
```

Este es el resultado:

```
<?xml version="1.0" encoding="utf-16" standalone="yes"?>
<test>data</test>
```

Observa que tenemos UTF-16 en la salida, a pesar de que solicitamos explícitamente UTF-8 en un XDeclaration. Esto puede parecer un error, pero de hecho, XmlWriter está siendo notablemente inteligente. Debido a que estamos escribiendo en una cadena de texto y no en un archivo o flujo, es imposible aplicar cualquier codificación que no sea UTF-16, que es el formato en el que se almacenan internamente las cadenas. Por lo tanto, XmlWriter escribe "utf-16" para no mentir.

Esto también explica por qué el método ToString no emite una declaración XML. Imagina que en lugar de llamar a Save, hicieras lo siguiente para escribir un XDocument en un archivo:

```
File.WriteAllText ("data.xml", doc.ToString());
```

Como está, data.xml carecería de una declaración XML, lo que lo haría incompleto pero aún parseable (se puede inferir la codificación de texto). Pero si ToString() emitiera una declaración XML, data.xml en realidad contendría una declaración incorrecta (encoding="utf-16"), lo que podría evitar que se lea en absoluto porque WriteAllText codifica utilizando UTF-8.

Nombres y espacios de nombres

Al igual que los tipos de .NET pueden tener espacios de nombres, también los elementos y atributos XML pueden tenerlos. Los espacios de nombres XML logran dos cosas. En primer lugar, al igual que los espacios de nombres en C#, ayudan a evitar colisiones de nombres. Esto puede convertirse en un problema cuando fusionas datos de un archivo XML en otro. En segundo lugar, los espacios de nombres asignan un significado absoluto a un nombre. El nombre "nil", por ejemplo, podría significar cualquier cosa. Sin embargo, dentro del espacio de nombres http://www.w3.org/2001/xmlschema-instance, "nil" significa algo equivalente a null en C# y viene con reglas específicas sobre cómo se puede aplicar.

Debido a que los espacios de nombres XML son una fuente significativa de confusión, primero cubrimos los espacios de nombres en general y luego pasamos a cómo se utilizan en LINQ to XML.

Espacios de nombres en XML

Supongamos que queremos definir un elemento de cliente en el espacio de nombres OReilly.Nutshell.CSharp. Hay dos formas de proceder. La primera es usar el atributo xmlns:

```
<customer xmlns="OReilly.Nutshell.CSharp"/>
```

xmlns es un atributo especial reservado. Cuando se usa de esta manera, realiza dos funciones:

- Especifica un espacio de nombres para el elemento en cuestión.
- Especifica un espacio de nombres predeterminado para todos los elementos descendientes.

Esto significa que en el siguiente ejemplo, address y postcode residen implícitamente en el espacio de nombres OReilly.Nutshell.CSharp:

```
<customer xmlns="OReilly.Nutshell.CSharp">
  <address>
    <postcode>02138</postcode>
  </address>
</customer>
```

Si queremos que "address" y "postcode" no tengan un espacio de nombres, tendríamos que hacer lo siguiente:

```
<customer xmlns="OReilly.Nutshell.CSharp">
  <address xmlns="">
    <postcode>02138</postcode>     <!-- postcode now inherits empty ns -->
  </address>
</customer>
```

Prefijos

La otra forma de especificar un espacio de nombres es con un prefijo. Un prefijo es un alias que asignas a un espacio de nombres para ahorrar escritura. Hay dos pasos para usar un prefijo: definir el prefijo y usarlo. Puedes hacer ambas cosas juntas:

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp"/>
```

Aquí están sucediendo dos cosas distintas. A la derecha, `xmlns:nut="..."` define un prefijo llamado "nut" y lo hace disponible para este elemento y todos sus descendientes. A la izquierda, `nut:customer` asigna el prefijo recién asignado al elemento "customer".

Un elemento con prefijo no define un espacio de nombres predeterminado para los descendientes. En el siguiente XML, "firstname" tiene un espacio de nombres vacío:

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp">
  <firstname>Joe</firstname>
</customer>
```

Para darle al "firstname" el prefijo "OReilly.Nutshell.CSharp", debes hacer esto:

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp">
  <nut:firstname>Joe</firstname>
</customer>
```

También puedes definir un prefijo (o varios) para la conveniencia de tus descendientes sin asignar ninguno de ellos al elemento padre en sí. Lo siguiente define dos prefijos, "i" y "z", mientras deja que el elemento "customer" en sí tenga un espacio de nombres vacío:

```
<customer xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  ...
</customer>
```

Si esto fuera el nodo raíz, todo el documento tendría acceso a los prefijos "i" y "z".

Los prefijos son convenientes cuando los elementos necesitan utilizar múltiples espacios de nombres. Observa que ambos espacios de nombres en este ejemplo son URIs. El uso de URIs (que posees) es una práctica estándar: garantiza la singularidad de los espacios de nombres. Entonces, en la vida real, nuestro elemento "customer" sería más probablemente así:

```
<customer xmlns="http://oreilly.com/schemas/nutshell/csharp"/>
```

O:

```
<nut:customer xmlns:nut="http://oreilly.com/schemas/nutshell/csharp"/>
```

Atributos

También se pueden asignar espacios de nombres a los atributos. La principal diferencia es que un atributo siempre requiere un prefijo. Por ejemplo:

```
<customer xmlns:nut="OReilly.Nutshell.CSharp" nut:id="123" />
```

Otra diferencia es que un atributo sin calificación siempre tiene un espacio de nombres vacío, nunca hereda un espacio de nombres predeterminado del elemento padre.

Los atributos generalmente no necesitan espacios de nombres porque su significado suele ser local al elemento. Una excepción son los atributos generales o de metadatos, como el atributo "nil" definido por W3C:

```
<customer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <firstname>Joe</firstname>
  <lastname xsi:nil="true"/>
</customer>
```

Esto indica de manera inequívoca que el apellido (lastname) es nulo (null en C#) y no una cadena vacía. Al utilizar el espacio de nombres estándar, una utilidad de análisis de propósito general puede comprender con certeza nuestra intención.

Especificación de espacios de nombres en X-DOM

Hasta ahora en este capítulo, hemos utilizado cadenas simples para los nombres de XElement y XAttribute. Una cadena simple corresponde a un nombre XML con un espacio de nombres vacío, similar a un tipo .NET definido en el espacio de nombres global.

Existen un par de formas de especificar un espacio de nombres XML. La primera es encerrarlo entre llaves, antes del nombre local:

```
var e = new XElement ("{http://domain.com/xmlspace}customer", "Bloggs");
Console.WriteLine (e.ToString());
```

Esto produce el XML resultante:

```
<customer xmlns="http://domain.com/xmlspace">Bloggs</customer>
```

La segunda forma (y más eficiente) es utilizar los tipos XNamespace y XName. Aquí están sus definiciones:

```
public sealed class XNamespace
{
  public string NamespaceName { get; }
}

public sealed class XName     // A local name with optional namespace
{
  public string LocalName { get; }
  public XNamespace Namespace { get; }   // Optional
}
```

Ambos tipos definen conversiones implícitas desde cadenas, por lo que lo siguiente es válido:

```
XNamespace ns   = "http://domain.com/xmlspace";
XName localName = "customer";
XName fullName  = "{http://domain.com/xmlspace}customer";
```

XNamespace también sobrecarga el operador +, lo que te permite combinar un espacio de nombres y un nombre en un XName sin utilizar llaves:

```
XNamespace ns = "http://domain.com/xmlspace";
XName fullName = ns + "customer";
Console.WriteLine (fullName);     // {http://domain.com/xmlspace}customer
```

Todos los constructores y métodos en el X-DOM que aceptan un nombre de elemento o atributo realmente aceptan un objeto XName en lugar de una cadena. La razón por la que puedes sustituir una cadena, como en todos nuestros ejemplos hasta ahora, es debido a la conversión implícita.

Especificar un espacio de nombres es igual, ya sea para un elemento o un atributo:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
              new XAttribute (ns + "id", 123)
           );
```

El X-DOM y los espacios de nombres predeterminados

El X-DOM ignora el concepto de espacios de nombres predeterminados hasta que llega el momento de generar XML. Esto significa que cuando construyes un XElement hijo, debes proporcionarle un espacio de nombres explícitamente si es necesario; no heredará el espacio de nombres del elemento padre:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement (ns + "customer", "Bloggs"),
             new XElement (ns + "purchase", "Bicycle")
           );
```

Sin embargo, el X-DOM aplica espacios de nombres predeterminados al leer y generar XML:

```
Console.WriteLine (data.ToString());

OUTPUT:
  <data xmlns="http://domain.com/xmlspace">
    <customer>Bloggs</customer>
    <purchase>Bicycle</purchase>
  </data>

Console.WriteLine (data.Element (ns + "customer").ToString());

OUTPUT:
  <customer xmlns="http://domain.com/xmlspace">Bloggs</customer>
```

Si construyes elementos XElement sin especificar espacios de nombres, es decir:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement ("customer", "Bloggs"),
             new XElement ("purchase", "Bicycle")


           );
Console.WriteLine (data.ToString());
```

obtendrás el siguiente resultado en su lugar:

```
<data xmlns="http://domain.com/xmlspace">
  <customer xmlns="">Bloggs</customer>
  <purchase xmlns="">Bicycle</purchase>
</data>
```

Otro error común es no incluir un espacio de nombres al navegar por un X-DOM:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement (ns + "customer", "Bloggs"),
             new XElement (ns + "purchase", "Bicycle")
           );
XElement x = data.Element (ns + "customer");    // ok
XElement y = data.Element ("customer");         // null
```

Si construyes un árbol X-DOM sin especificar espacios de nombres, puedes asignar posteriormente todos los elementos a un único espacio de nombres de la siguiente manera:

```
foreach (XElement e in data.DescendantsAndSelf())
  if (e.Name.Namespace == "")
    e.Name = ns + e.Name.LocalName;
```

Prefijos

El X-DOM trata los prefijos de la misma manera que trata los espacios de nombres: simplemente como una función de serialización. Esto significa que puedes optar por ignorar completamente el tema de los prefijos y aún así funcionar. La única razón por la que podrías querer hacer lo contrario es por eficiencia al generar un archivo XML. Por ejemplo, considera lo siguiente:

```
XNamespace ns1 = "http://domain.com/space1";
XNamespace ns2 = "http://domain.com/space2";

var mix = new XElement (ns1 + "data",
            new XElement (ns2 + "element", "value"),
            new XElement (ns2 + "element", "value"),
            new XElement (ns2 + "element", "value")
          );
```

De forma predeterminada, XElement serializará esto de la siguiente manera:

```
<data xmlns="http://domain.com/space1">
  <element xmlns="http://domain.com/space2">value</element>
  <element xmlns="http://domain.com/space2">value</element>
  <element xmlns="http://domain.com/space2">value</element>
</data>
```

Como puedes ver, hay un poco de duplicación innecesaria. La solución no es cambiar la forma en que construyes el X-DOM, sino proporcionar pistas al serializador antes de escribir el XML. Haz esto agregando atributos que definan los prefijos que deseas ver aplicados. Esto se hace típicamente en el elemento raíz:

```
mix.SetAttributeValue (XNamespace.Xmlns + "ns1", ns1);
mix.SetAttributeValue (XNamespace.Xmlns + "ns2", ns2);
```

Esto asigna el prefijo "ns1" a nuestra variable XNamespace ns1 y "ns2" a ns2. El X-DOM automáticamente recoge estos atributos al serializar y los utiliza para condensar el XML resultante. Aquí está el resultado ahora de llamar a ToString en mix:

```
<ns1:data xmlns:ns1="http://domain.com/space1"
          xmlns:ns2="http://domain.com/space2">
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
</ns1:data>
```

Los prefijos no cambian la forma en que construyes, consultas o actualizas el X-DOM, para estas actividades, ignoras la presencia de los prefijos y continúas usando nombres completos. Los prefijos entran en juego solo al convertir desde y hacia archivos o flujos XML.

Los prefijos también se respetan al serializar atributos. En el siguiente ejemplo, registramos la fecha de nacimiento y el crédito de un cliente como "nil" utilizando el atributo estándar de W3C. La línea resaltada asegura que el prefijo se serialice sin repetición innecesaria de espacio de nombres:

```
XNamespace xsi = "http://www.w3.org/2001/XMLSchema-instance";
var nil = new XAttribute (xsi + "nil", true);

var cust = new XElement ("customers",
             new XAttribute (XNamespace.Xmlns + "xsi", xsi),
             new XElement ("customer",
               new XElement ("lastname", "Bloggs"),
               new XElement ("dob", nil),
               new XElement ("credit", nil)
             )
           );
```

Este es su XML:

```
<customers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <customer>
    <lastname>Bloggs</lastname>
    <dob xsi:nil="true" />
    <credit xsi:nil="true" />
  </customer>
</customers>
```

Para mayor concisión, predeclaramos el XAttribute nil para poder usarlo dos veces en la construcción del DOM. Se te permite hacer referencia al mismo atributo dos veces porque se duplica automáticamente según sea necesario.

Anotaciones

Puedes adjuntar datos personalizados a cualquier XObject con una anotación. Las anotaciones están destinadas para tu uso privado y son tratadas como cajas negras por el X-DOM. Si alguna vez has utilizado la propiedad Tag en un control de Windows Forms o Windows Presentation Foundation (WPF), estarás familiarizado con el concepto, la diferencia es que tienes múltiples anotaciones y tus anotaciones pueden tener un ámbito privado.

Puedes crear una anotación que otros tipos ni siquiera pueden ver, y mucho menos sobrescribir. Los siguientes métodos en XObject agregan y eliminan anotaciones:

```
public void AddAnnotation (object annotation)
public void RemoveAnnotations<T>()     where T : class
```

Los siguientes métodos recuperan anotaciones:

```
public T Annotation<T>()               where T : class
public IEnumerable<T> Annotations<T>() where T : class
```

Cada anotación se identifica por su tipo, que debe ser un tipo de referencia. El siguiente ejemplo agrega y luego recupera una anotación de tipo string:

```
XElement e = new XElement ("test");
e.AddAnnotation ("Hello");
Console.WriteLine (e.Annotation<string>());   // Hello
```

Puedes agregar múltiples anotaciones del mismo tipo y luego usar el método Annotations para recuperar una secuencia de coincidencias.

Sin embargo, un tipo público como string no es la mejor opción como clave, ya que el código en otros tipos puede interferir con tus anotaciones. Un enfoque mejor es utilizar una clase interna o (anidada) privada:

```
class X
{
  class CustomData { internal string Message; }   // Private nested type

  static void Test()
  {
    XElement e = new XElement ("test");
    e.AddAnnotation (new CustomData { Message = "Hello" } );
    Console.Write (e.Annotations<CustomData>().First().Message);  // Hello
  }
}
```

Para eliminar anotaciones, también debes tener acceso al tipo de clave:

```
e.RemoveAnnotations<CustomData>();
```

Proyección en un X-DOM

Hasta ahora, hemos mostrado cómo utilizar LINQ para obtener datos de un X-DOM. También puedes utilizar consultas LINQ para proyectar en un X-DOM. La fuente puede ser cualquier cosa sobre la cual se pueda hacer una consulta LINQ, como las siguientes:

- Clases de entidad de EF Core
- Una colección local
- Otro X-DOM

Independientemente de la fuente, la estrategia es la misma al utilizar LINQ para generar un X-DOM: primero se escribe una expresión de construcción funcional que produce la forma deseada del X-DOM y luego se crea una consulta LINQ alrededor de la expresión.

Por ejemplo, supongamos que queremos recuperar clientes de una base de datos y generar el siguiente XML:

```
<customers>
  <customer id="1">
    <name>Sue</name>
    <buys>3</buys>
  </customer>
  ...
</customers>
```

Comenzamos escribiendo una expresión de construcción funcional para el X-DOM utilizando literales simples:

```
var customers =
  new XElement ("customers",
    new XElement ("customer", new XAttribute ("id", 1),
      new XElement ("name", "Sue"),
      new XElement ("buys", 3)
    )
  );
```

Luego convertimos esto en una proyección y creamos una consulta LINQ alrededor de ella:

```
var customers =
  new XElement ("customers",
    // We must call AsEnumerable() due to a bug in EF Core.
    from c in dbContext.Customers.AsEnumerable()
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count)
      )
    );
```



La llamada a AsEnumerable es necesaria debido a un error en EF Core (se planea una solución en una versión posterior). Después de que se corrija el error, eliminar la llamada a AsEnumerable mejorará la eficiencia al evitar una ida y vuelta en cada llamada a c.Purchases.Count.

Aquí está el resultado:

```
<customers>
  <customer id="1">
    <name>Tom</name>
    <buys>3</buys>
  </customer>
  <customer id="2">
    <name>Harry</name>
    <buys>2</buys>
  </customer>

    ...
</customers>
```

Podemos ver cómo esto funciona de manera más clara construyendo la misma consulta en dos pasos. Primero:

```
IEnumerable<XElement> sqlQuery =
  from c in dbContext.Customers.AsEnumerable()
  select
    new XElement ("customer", new XAttribute ("id", c.ID),
      new XElement ("name", c.Name),
      new XElement ("buys", c.Purchases.Count)
    );
```

Esta parte interna es una consulta LINQ normal que proyecta en XElements. Aquí está el segundo paso:

```
var customers = new XElement ("customers", sqlQuery);
```

Esto construye el XElement raíz. Lo único inusual es que el contenido, sqlQuery, no es un solo XElement, sino un IQueryable<XElement>, que implementa IEnumerable<XElement>. Recuerda que en el procesamiento de contenido XML, las colecciones se enumeran automáticamente. Por lo tanto, cada XElement se agrega como un nodo secundario.

Eliminar elementos vacíos

Supongamos que en el ejemplo anterior también queremos incluir detalles de la compra de alto valor más reciente del cliente. Podemos hacerlo de la siguiente manera:

```
var customers =
  new XElement ("customers",
    // The AsEnumerable call can be removed when the EF Core bug is fixe
    from c in dbContext.Customers.AsEnumerable()
    let lastBigBuy = (from p in c.Purchases
                      where p.Price > 1000
                      orderby p.Date descending
                      select p).FirstOrDefault()
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count),
        new XElement ("lastBigBuy",
          new XElement ("description", lastBigBuy?.Description),
          new XElement ("price", lastBigBuy?.Price ?? 0m)
        )
      )
  );
```

Esto emite elementos vacíos para los clientes que no tienen compras de alto valor. (Si fuera una consulta local en lugar de una consulta de base de datos, lanzaría una excepción NullReferenceException). En tales casos, sería mejor omitir por completo el nodo lastBigBuy. Podemos lograr esto envolviendo el constructor del elemento lastBigBuy en un operador condicional:

```
select
  new XElement ("customer", new XAttribute ("id", c.ID),
    new XElement ("name", c.Name),
    new XElement ("buys", c.Purchases.Count),
    lastBigBuy == null ? null :
      new XElement ("lastBigBuy",
        new XElement ("description", lastBigBuy.Description),
        new XElement ("price", lastBigBuy.Price)
```

Para clientes sin `lastBigBuy`, se emite un valor nulo en lugar de un XElement vacío. Esto es lo que queremos, ya que el contenido nulo simplemente se ignora.

Transmitiendo una proyección

Si estás proyectando en un X-DOM solo para guardarlo (o llamar a ToString), puedes mejorar la eficiencia de memoria utilizando un XStreamingElement. Un XStreamingElement es una versión simplificada de XElement que aplica la carga diferida a su contenido hijo. Para usarlo, simplemente reemplazas los XElement externos por XStreamingElements:

```
var customers =
  new XStreamingElement ("customers",
    from c in dbContext.Customers
    select
      new XStreamingElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count)
      )
    );
customers.Save ("data.xml");
```

Las consultas pasadas al constructor de XStreamingElement no se enumeran hasta que llamas a Save, ToString o WriteTo en el elemento; esto evita cargar todo el X-DOM en memoria de una vez. La contrapartida es que las consultas se vuelven a evaluar si vuelves a guardar. Además, no puedes recorrer el contenido hijo de un XStreamingElement, ya que no expone métodos como Elements o Attributes.

XStreamingElement no se basa en XObject ni en ninguna otra clase, debido a su conjunto limitado de miembros. Los únicos miembros que tiene, además de Save, ToString y WriteTo, son los siguientes:

- Un método Add, que acepta contenido similar al constructor.
- Una propiedad Name.

XStreamingElement no te permite leer contenido de forma transmitida, para esto debes usar un XmlReader en conjunto con el X-DOM. Describimos cómo hacer esto en "Patrones para usar XmlReader/XmlWriter" en la página 539.

11 Otras tecnologías XML y JSON

En el Capítulo 10, cubrimos la API LINQ-to-XML y XML en general. En este capítulo, exploramos las clases XmlReader/XmlWriter de bajo nivel y los tipos para trabajar con JavaScript Object Notation (JSON), que se ha convertido en una alternativa popular a XML.

En el suplemento en línea, describimos las herramientas para trabajar con esquemas XML y hojas de estilo.

XmlReader

XmlReader es una clase de alto rendimiento para leer un flujo XML de manera eficiente y en un solo sentido.

Considera el siguiente archivo XML, customer.xml:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

Para instanciar un XmlReader, se utiliza el método estático XmlReader.Create, pasando un Stream, un TextReader o una cadena de URI:

```
using XmlReader reader = XmlReader.Create ("customer.xml");
  ...
```



Debido a que XmlReader te permite leer desde fuentes potencialmente lentas (Streams y URIs), ofrece versiones asíncronas de la mayoría de sus métodos para que puedas escribir fácilmente código no bloqueante.

Exploramos la asincronía en detalle en el Capítulo 14.

Para construir un XmlReader que lea desde una cadena, puedes utilizar el siguiente código:

```
using XmlReader reader = XmlReader.Create (
  new System.IO.StringReader (myString));
```

También puedes pasar un objeto XmlReaderSettings para controlar las opciones de análisis y validación. Las siguientes tres propiedades en XmlReaderSettings son particularmente útiles para omitir contenido innecesario:

```
bool IgnoreComments                 // Skip over comment nodes?
bool IgnoreProcessingInstructions   // Skip over processing instructions?
bool IgnoreWhitespace               // Skip over whitespace?
```

En el siguiente ejemplo, configuramos el lector para que no emita nodos de espacios en blanco, los cuales pueden ser una distracción en escenarios típicos:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using XmlReader reader = XmlReader.Create ("customer.xml", settings);
  ...
```

Otra propiedad útil en XmlReaderSettings es ConformanceLevel. Su valor predeterminado, Document, indica al lector que asuma un documento XML válido con un único nodo raíz. Esto puede ser problemático si deseas leer solo una parte interna del XML que contiene múltiples nodos, como en el siguiente ejemplo:

```
<firstname>Jim</firstname>
<lastname>Bo</lastname>
```

Para leer esto sin lanzar una excepción, debes establecer ConformanceLevel en Fragment.

XmlReaderSettings también tiene una propiedad llamada CloseInput, que indica si se debe cerrar el flujo subyacente cuando se cierra el lector (hay una propiedad análoga en XmlWriterSettings llamada CloseOutput). El valor predeterminado de CloseInput y CloseOutput es false.

Lectura de nodos

Las unidades de un flujo XML son los nodos XML. El lector recorre el flujo en orden textual (en orden de profundidad). La propiedad Depth del lector devuelve la profundidad actual del cursor.

La forma más primitiva de leer desde un XmlReader es llamar a Read. Avanza al siguiente nodo en el flujo XML, de manera similar a MoveNext en IEnumerator. La primera llamada a Read posiciona el cursor en el primer nodo. Cuando Read devuelve false, significa que el cursor ha avanzado más allá del último nodo y en ese punto se debe cerrar y abandonar el XmlReader.

Dos propiedades de tipo string en XmlReader proporcionan acceso al contenido de un nodo: Name y Value. Dependiendo del tipo de nodo, se completará Name o Value (o ambos).

En este ejemplo, leemos cada nodo en el flujo XML, mostrando el tipo de cada nodo a medida que avanzamos:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using XmlReader reader = XmlReader.Create ("customer.xml", settings);
while (reader.Read())
{
  Console.Write (new string (' ', reader.Depth * 2));  // Write indentation
  Console.Write (reader.NodeType.ToString());

  if (reader.NodeType == XmlNodeType.Element ||
      reader.NodeType == XmlNodeType.EndElement)
  {
    Console.Write (" Name=" + reader.Name);
  }
  else if (reader.NodeType == XmlNodeType.Text)
  {
    Console.Write (" Value=" + reader.Value);
  }
  Console.WriteLine ();
}
```

La salida sería la siguiente:

```
XmlDeclaration
Element Name=customer
  Element Name=firstname
    Text Value=Jim
  EndElement Name=firstname
  Element Name=lastname
    Text Value=Bo
  EndElement Name=lastname
EndElement Name=customer
```



Los atributos no se incluyen en el recorrido basado en Read (consulta "Lectura de atributos" en la página 535).

NodeType es del tipo XmlNodeType, que es una enumeración con los siguientes miembros:

None	Comment	Document
XmlDeclaration	Entity	DocumentType
Element	EndEntity	DocumentFragment
EndElement	EntityReference	Notation
Text	ProcessingInstruction	Whitespace
Attribute	CDATA	SignificantWhitespace

Lectura de elementos

A menudo, ya conoces la estructura del documento XML que estás leyendo. Para ayudarte con esto, XmlReader proporciona una serie de métodos que leen asumiendo una estructura particular. Esto simplifica tu código y realiza algunas validaciones al mismo tiempo.



XmlReader lanza una excepción XmlException si falla alguna validación.

XmlException tiene las propiedades LineNumber y LinePosition que indican dónde ocurrió el error. Registrar esta información es esencial si el archivo XML es grande.

ReadStartElement verifica que el NodeType actual sea Element y luego llama a Read. Si especificas un nombre, verifica que coincida con el del elemento actual.

ReadEndElement verifica que el NodeType actual sea EndElement y luego llama a Read.

Por ejemplo, podríamos leer esto:

```
<firstname>Jim</firstname>
```

de la siguiente manera:

```
reader.ReadStartElement ("firstname");
Console.WriteLine (reader.Value);
reader.Read();
reader.ReadEndElement();
```

El método ReadElementContentAsString hace todo esto de una vez. Lee un elemento de inicio, un nodo de texto y un elemento de cierre, y devuelve el contenido como una cadena:

```
string firstName = reader.ReadElementContentAsString ("firstname", "");
```

El segundo argumento se refiere al espacio de nombres, que en este ejemplo está en blanco.

También existen versiones con tipos específicos de este método, como ReadElementContentAsInt, que realizan el análisis del resultado. Volviendo a nuestro documento XML original:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <creditlimit>500.00</creditlimit>    <!-- OK, we sneaked this in! -->
</customer>
```

Podríamos leerlo de la siguiente manera:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using XmlReader r = XmlReader.Create ("customer.xml", settings);

r.MoveToContent();                // Skip over the XML declaration
r.ReadStartElement ("customer");
string firstName    = r.ReadElementContentAsString ("firstname", "");
string lastName     = r.ReadElementContentAsString ("lastname", "");
decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");

r.MoveToContent();      // Skip over that pesky comment
r.ReadEndElement();     // Read the closing customer tag
```



El método MoveToContent es realmente útil. Omite todo lo innecesario: declaraciones XML, espacios en blanco, comentarios e instrucciones de procesamiento. También puedes indicarle al lector que realice la mayoría de estas omisiones automáticamente a través de las propiedades en XmlReaderSettings.

Elementos opcionales

En el ejemplo anterior, supongamos que <lastname> es opcional. La solución a esto es sencilla:

```
r.ReadStartElement ("customer");
string firstName    = r. ReadElementContentAsString ("firstname", "");
string lastName     = r.Name == "lastname"
                      ? r.ReadElementContentAsString() : null;
decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");
```

Orden aleatoria de elementos

Los ejemplos en esta sección dependen de que los elementos aparezcan en el archivo XML en un orden establecido. Si necesitas manejar elementos que aparecen en cualquier orden, la solución más fácil es leer esa sección del XML en un X-DOM. Describimos cómo hacer esto más adelante en "Patrones para usar XmlReader/XmlWriter" en la página 539.

Elementos vacíos

La forma en que XmlReader maneja los elementos vacíos presenta un problema. Considera el siguiente elemento:

```
<customerList></customerList>
```

En XML, esto es equivalente a:

```
<customerList/>
```

Sin embargo, XmlReader trata los dos casos de manera diferente. En el primer caso, el siguiente código funciona como se espera:

```
reader.ReadStartElement ("customerList");
reader.ReadEndElement();
```

En el segundo caso, ReadEndElement lanza una excepción porque no hay un "elemento de cierre" separado según XmlReader. La solución es verificar si el elemento está vacío:

```
bool isEmpty = reader.IsEmptyElement;
reader.ReadStartElement ("customerList");
if (!isEmpty) reader.ReadEndElement();
```

En realidad, esto solo es molesto cuando el elemento en cuestión podría contener elementos secundarios (como una lista de clientes). Con elementos que envuelven texto simple (como firstname), puedes evitar todo el problema llamando a un método como ReadElementContentAsString. Los métodos ReadElementXXX manejan correctamente ambos tipos de elementos vacíos.

Otros métodos ReadXXX

La Tabla 11-1 resume todos los métodos ReadXXX en XmlReader. La mayoría de ellos están diseñados para trabajar con elementos. El fragmento XML de muestra mostrado en negrita es la sección leída por el método descrito.

Table 11-1. Read methods

Members	Works on NodeType	Sample XML fragment	Input parameters	Data returned
ReadContentAs*XXX*	Text	<a>**x**		x
ReadElement ContentAs*XXX*	Element	**<a>x**		x
ReadInnerXml	Element	**<a>x**		x
ReadOuterXml	Element	**<a>x**		<a>x
ReadStartElement	Element	**<a>**x		
ReadEndElement	Element	<a>x****		
ReadSubtree	Element	**<a>x**		<a>x
ReadToDescendant	Element	**<a>x**	"b"	
ReadToFollowing	Element	**<a>x**	"b"	
ReadToNextSibling	Element	**<a>x**	"b"	
ReadAttribute Value	Attribute	See "Reading Attributes" on page 535		

Los métodos ReadContentAsXXX analizan un nodo de texto en el tipo XXX. Internamente, la clase XmlConvert realiza la conversión de cadena a tipo. El nodo de texto puede estar dentro de un elemento o un atributo.

Los métodos ReadElementContentAsXXX son envoltorios alrededor de los correspondientes métodos ReadContentAsXXX. Se aplican al nodo de elemento en lugar del nodo de texto contenido por el elemento.

ReadInnerXml se aplica típicamente a un elemento y lee y devuelve un elemento y todos sus descendientes. Cuando se aplica a un atributo, devuelve el valor del atributo. ReadOuterXml es similar, excepto que incluye en lugar de excluir el elemento en la posición del cursor.

ReadSubtree devuelve un lector proxy que proporciona una vista solo del elemento actual (y sus descendientes). El lector proxy debe cerrarse antes de que el lector original pueda ser leído nuevamente de forma segura. Cuando se cierra el lector proxy, la posición del cursor del lector original se mueve al final del subárbol.

ReadToDescendant mueve el cursor al inicio del primer nodo descendiente con el nombre/espacio de nombres especificado. ReadToFollowing mueve el cursor al inicio del primer nodo, sin importar la profundidad, con el nombre/espacio de nombres especificado. ReadToNextSibling mueve el cursor al inicio del primer nodo hermano con el nombre/espacio de nombres especificado.

También existen dos métodos heredados: ReadString y ReadElementString se comportan como ReadContentAsString y ReadElementContentAsString, excepto que lanzan una excepción si hay más de un nodo de texto dentro del elemento. Debes evitar estos métodos porque lanzan una excepción si un elemento contiene un comentario.

Lectura de atributos

XmlReader proporciona un indexador que te permite acceder directamente (de manera aleatoria) a los atributos de un elemento, ya sea por nombre o por posición. Usar el indexador es equivalente a llamar a GetAttribute.

Dado el fragmento XML:

```
<customer id="123" status="archived"/>
```

podríamos leer sus atributos de la siguiente manera:

```
Console.WriteLine (reader ["id"]);              // 123
Console.WriteLine (reader ["status"]);          // archived
Console.WriteLine (reader ["bogus"] == null);   // True
```

 El XmlReader debe estar posicionado en un elemento de inicio para poder leer los atributos. ¡Después de llamar a ReadStartElement, los atributos desaparecen para siempre!

Aunque el orden de los atributos no tiene relevancia semántica, puedes acceder a los atributos por su posición ordinal. Podríamos reescribir el ejemplo anterior de la siguiente manera:

```
Console.WriteLine (reader [0]);           // 123
Console.WriteLine (reader [1]);           // archived
```

El indexador también te permite especificar el espacio de nombres del atributo, si lo tiene.

AttributeCount devuelve el número de atributos del nodo actual.

Nodos de atributos

Para recorrer explícitamente los nodos de atributos, debes hacer una desviación especial del camino normal de simplemente llamar a Read. Una buena razón para hacerlo es si deseas analizar los valores de los atributos en otros tipos mediante los métodos ReadContentAsXXX.

La desviación debe comenzar desde un elemento de inicio. Para facilitar el trabajo, la regla de solo avance se relaja durante el recorrido de los atributos: puedes saltar a cualquier atributo (hacia adelante o hacia atrás) llamando a MoveToAttribute.



MoveToElement te devuelve al elemento de inicio desde cualquier lugar dentro de la desviación del nodo de atributo.

Retomando nuestro ejemplo anterior:

```
<customer id="123" status="archived"/>
```

podemos hacer lo siguiente:

```
reader.MoveToAttribute ("status");
string status = reader.ReadContentAsString();

reader.MoveToAttribute ("id");
int id = reader.ReadContentAsInt();
```

MoveToAttribute devuelve false si el atributo especificado no existe.

También puedes recorrer cada atributo en secuencia llamando a los métodos MoveToFirstAttribute y luego MoveToNextAttribute:

```
if (reader.MoveToFirstAttribute())
  do { Console.WriteLine (reader.Name + "=" + reader.Value);  }
  while (reader.MoveToNextAttribute());

// OUTPUT:
id=123
status=archived
```

Espacios de nombres y prefijos

XmlReader proporciona dos sistemas paralelos para referirse a los nombres de elementos y atributos:

- Name (Nombre)
- NamespaceURI y LocalName (Espacio de nombres y Nombre local)

Cuando lees la propiedad Name de un elemento o llamas a un método que acepta un único argumento de nombre, estás utilizando el primer sistema. Esto funciona bien si no hay espacios de nombres ni prefijos presentes; de lo contrario, actúa de manera básica y literal. Los espacios de nombres se ignoran y los prefijos se incluyen exactamente como se escribieron. Por ejemplo:

Sample fragment	Name
`<customer ...>`	customer
`<customer xmlns='blah' ...>`	customer
`<x:customer ...>`	x:customer

El siguiente código funciona para los primeros dos casos:

```
reader.ReadStartElement ("customer");
```

Lo siguiente es necesario para manejar el tercer caso:

```
reader.ReadStartElement ("x:customer");
```

El segundo sistema funciona a través de dos propiedades que tienen en cuenta los espacios de nombres: NamespaceURI y LocalName. Estas propiedades consideran los prefijos y los espacios de nombres predeterminados definidos por los elementos padre. Los prefijos se expanden automáticamente. Esto significa que NamespaceURI siempre refleja el espacio de nombres semánticamente correcto para el elemento actual, y LocalName siempre está libre de prefijos.

Cuando pasas dos argumentos de nombre a un método como ReadStartElement, estás utilizando este mismo sistema. Por ejemplo, considera el siguiente XML:

```
<customer xmlns="DefaultNamespace" xmlns:other="OtherNamespace">
  <address>
    <other:city>
    ...
```

Podemos leer esto de la siguiente manera:

```
reader.ReadStartElement ("customer", "DefaultNamespace");
reader.ReadStartElement ("address",  "DefaultNamespace");
reader.ReadStartElement ("city",     "OtherNamespace");
```

Abstraer los prefijos es generalmente lo que deseas. Si es necesario, puedes ver qué prefijo se usó a través de la propiedad Prefix y convertirlo en un espacio de nombres llamando a LookupNamespace.

XmlWriter

XmlWriter es un escritor de flujo XML de un solo sentido. El diseño de XmlWriter es simétrico al de XmlReader.

Al igual que con XmlTextReader, construyes un XmlWriter llamando a Create con un objeto de configuración opcional. En el siguiente ejemplo, habilitamos la indentación para que la salida sea más legible y luego escribimos un archivo XML simple:

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;

using XmlWriter writer = XmlWriter.Create ("foo.xml", settings);

writer.WriteStartElement ("customer");
writer.WriteElementString ("firstname", "Jim");
writer.WriteElementString ("lastname", "Bo");
writer.WriteEndElement();
```

Esto produce el siguiente documento (el mismo que el archivo que leímos en el primer ejemplo de XmlReader):

```
<?xml version="1.0" encoding="utf-8"?>
<customer>
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

XmlWriter escribe automáticamente la declaración al principio a menos que indiques lo contrario en XmlWriterSettings estableciendo OmitXmlDeclaration en true o ConformanceLevel en Fragment. Este último también permite escribir múltiples nodos raíz, algo que de otro modo generaría una excepción.

El método WriteValue escribe un único nodo de texto. Acepta tanto tipos de cadena como tipos que no son de cadena, como bool y DateTime, llamando internamente a XmlConvert para realizar conversiones de cadena compatibles con XML:

```
writer.WriteStartElement ("birthdate");
writer.WriteValue (DateTime.Now);
writer.WriteEndElement();
```

En contraste, si llamamos a

```
WriteElementString ("birthdate", DateTime.Now.ToString());
```

el resultado no sería compatible con XML y podría ser vulnerable a una interpretación incorrecta.

WriteString es equivalente a llamar a WriteValue con una cadena. XmlWriter escapa automáticamente los caracteres que de otro modo serían ilegales dentro de un atributo o elemento, como &, < > y caracteres Unicode extendidos.

Escritura de atributos

Puedes escribir atributos inmediatamente después de escribir un elemento de inicio:

```
writer.WriteStartElement ("customer");
writer.WriteAttributeString ("id", "1");
writer.WriteAttributeString ("status", "archived");
```

Para escribir valores que no son de cadena, llama a WriteStartAttribute, WriteValue y luego a WriteEndAttribute.

Escritura de otros tipos de nodos

XmlWriter también define los siguientes métodos para escribir otros tipos de nodos:

```
WriteBase64       // for binary data
WriteBinHex       // for binary data
WriteCData
WriteComment
WriteDocType
WriteEntityRef
WriteProcessingInstruction
WriteRaw
WriteWhitespace
```

WriteRaw inyecta directamente una cadena en la secuencia de salida. También hay un método WriteNode que acepta un XmlReader y repite todo lo que se encuentra en el XmlReader dado.

Espacios de nombres y prefijos

Las sobrecargas de los métodos Write* te permiten asociar un elemento o atributo con un espacio de nombres. Reescribamos el contenido del archivo XML en nuestro ejemplo anterior. En esta ocasión, asociaremos todos los elementos con el espacio de nombres http://oreilly.com y declararemos el prefijo "o" en el elemento customer:

```
writer.WriteStartElement ("o", "customer", "http://oreilly.com");
writer.WriteElementString ("o", "firstname", "http://oreilly.com", "Jim");
writer.WriteElementString ("o", "lastname", "http://oreilly.com", "Bo");
writer.WriteEndElement();
```

El resultado ahora es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<o:customer xmlns:o='http://oreilly.com'>
  <o:firstname>Jim</o:firstname>
  <o:lastname>Bo</o:lastname>
</o:customer>
```

Observa cómo, por brevedad, XmlWriter omite las declaraciones de espacio de nombres de los elementos secundarios cuando ya han sido declarados por el elemento padre.

Patrones para usar XmlReader/XmlWriter

Trabajar con datos jerárquicos

Consideremos las siguientes clases:

```
public class Contacts
{
  public IList<Customer> Customers = new List<Customer>();
  public IList<Supplier> Suppliers = new List<Supplier>();
}

public class Customer { public string FirstName, LastName; }
public class Supplier { public string Name;                 }
```

Supongamos que quieres usar XmlReader y XmlWriter para serializar un objeto Contacts a XML, como se muestra a continuación:

```
<?xml version="1.0" encoding="utf-8"?>
<contacts>
   <customer id="1">
      <firstname>Jay</firstname>
      <lastname>Dee</lastname>
   </customer>
   <customer>                    <!-- we'll assume id is optional -->
      <firstname>Kay</firstname>
      <lastname>Gee</lastname>
   </customer>
   <supplier>
      <name>X Technologies Ltd</name>
   </supplier>
</contacts>
```

El enfoque más adecuado no es escribir un solo método grande, sino encapsular la funcionalidad XML en los propios tipos Customer y Supplier mediante la escritura de los métodos ReadXml y WriteXml en estos tipos. El patrón para hacerlo es sencillo:

- Los métodos ReadXml y WriteXml mantienen el lector/escritor a la misma profundidad cuando salen.
- ReadXml lee el elemento externo, mientras que WriteXml solo escribe su contenido interno.

Aquí está cómo escribiríamos el tipo Customer:

```
public class Customer
{
  public const string XmlName = "customer";
  public int? ID;
  public string FirstName, LastName;

  public Customer () { }
  public Customer (XmlReader r) { ReadXml (r); }

  public void ReadXml (XmlReader r)
  {
    if (r.MoveToAttribute ("id")) ID = r.ReadContentAsInt();
    r.ReadStartElement();
    FirstName = r.ReadElementContentAsString ("firstname", "");
    LastName = r.ReadElementContentAsString ("lastname", "");
    r.ReadEndElement();
  }

  public void WriteXml (XmlWriter w)
  {
    if (ID.HasValue) w.WriteAttributeString ("id", "", ID.ToString());
    w.WriteElementString ("firstname", FirstName);
    w.WriteElementString ("lastname", LastName);
  }
}
```

Observa que ReadXml lee los nodos de inicio y fin del elemento externo. Si su llamador hiciera este trabajo en su lugar, Customer no podría leer sus propios atributos. La razón por la que WriteXml no es simétrico en este sentido se debe a dos motivos:

- El llamador puede necesitar elegir el nombre del elemento externo.

- El llamador puede necesitar escribir atributos XML adicionales, como el subtipo del elemento (que luego podría usarse para decidir qué clase instanciar al leer de nuevo el elemento).

Otro beneficio de seguir este patrón es que hace que tu implementación sea compatible con IXmlSerializable (esto se cubre en "Serialización" en el suplemento en línea en http://www.albahari.com/nutshell).

La clase Supplier es análoga:

```
public class Supplier
{
  public const string XmlName = "supplier";
  public string Name;

  public Supplier () { }
  public Supplier (XmlReader r) { ReadXml (r); }

  public void ReadXml (XmlReader r)
  {
    r.ReadStartElement();
    Name = r.ReadElementContentAsString ("name", "");
    r.ReadEndElement();
  }

  public void WriteXml (XmlWriter w) =>
    w.WriteElementString ("name", Name);
}
```

Con la clase Contacts, debemos enumerar el elemento "customers" en ReadXml, comprobando si cada subelemento es un "customer" o un "supplier". También necesitamos lidiar con la trampa del elemento vacío:

```
public void ReadXml (XmlReader r)
{
  bool isEmpty = r.IsEmptyElement;           // This ensures we don't get
  r.ReadStartElement();                      // snookered by an empty
  if (isEmpty) return;                       // <contacts/> element!
  while (r.NodeType == XmlNodeType.Element)
  {
    if (r.Name == Customer.XmlName)      Customers.Add (new Customer (r));
    else if (r.Name == Supplier.XmlName) Suppliers.Add (new Supplier (r));
    else
      throw new XmlException ("Unexpected node: " + r.Name);
  }
  r.ReadEndElement();
}

public void WriteXml (XmlWriter w)
{
  foreach (Customer c in Customers)
  {
    w.WriteStartElement (Customer.XmlName);
    c.WriteXml (w);
    w.WriteEndElement();
  }
  foreach (Supplier s in Suppliers)
  {
    w.WriteStartElement (Supplier.XmlName);
    s.WriteXml (w);
    w.WriteEndElement();
  }
}
```

Aquí tienes cómo serializar un objeto Contacts lleno de Customers y Suppliers a un archivo XML:

```
var settings = new XmlWriterSettings();
settings.Indent = true;  // To make visual inspection easier

using XmlWriter writer = XmlWriter.Create ("contacts.xml", settings);

var cts = new Contacts()
// Add Customers and Suppliers...

writer.WriteStartElement ("contacts");
cts.WriteXml (writer);
writer.WriteEndElement();
```

Aquí tienes cómo deserializar desde el mismo archivo:

```
var settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
settings.IgnoreComments = true;
settings.IgnoreProcessingInstructions = true;

using XmlReader reader = XmlReader.Create("contacts.xml", settings);
reader.MoveToContent();
var cts = new Contacts();
cts.ReadXml(reader);
```

Combinar XmlReader/XmlWriter con un X-DOM

Puedes utilizar un X-DOM en cualquier punto del árbol XML donde XmlReader o XmlWriter se vuelvan demasiado complicados. Usar el X-DOM para manejar elementos internos es una excelente manera de combinar la facilidad de uso del X-DOM con el bajo consumo de memoria de XmlReader y XmlWriter.

Usando XmlReader con XElement

Para leer el elemento actual en un X-DOM, llamas a XNode.ReadFrom, pasando el XmlReader. A diferencia de XElement.Load, este método no es "codicioso" en el sentido de que no espera ver un documento completo. En su lugar, lee solo el final del subárbol actual.

Por ejemplo, supongamos que tenemos un archivo de registro XML estructurado de la siguiente manera:

```
<log>
  <logentry id="1">
    <date>...</date>
    <source>...</source>
    ...
  </logentry>
  ...
</log>
```

Si hubiera un millón de elementos logentry, leer todo en un X-DOM desperdiciaría memoria. Una solución mejor es recorrer cada logentry con un XmlReader y luego usar XElement para procesar los elementos individualmente:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using XmlReader r = XmlReader.Create ("logfile.xml", settings);

r.ReadStartElement ("log");
while (r.Name == "logentry")
{
  XElement logEntry = (XElement) XNode.ReadFrom (r);
  int id = (int) logEntry.Attribute ("id");
  DateTime date = (DateTime) logEntry.Element ("date");
  string source = (string) logEntry.Element ("source");
  ...
}
r.ReadEndElement();
```

Si sigues el patrón descrito en la sección anterior, puedes incluir un XElement en el método ReadXml o WriteXml de un tipo personalizado sin que el llamador se dé cuenta de que has hecho trampa. Por ejemplo, podríamos reescribir el método ReadXml de Customer de la siguiente manera:

```
public void ReadXml (XmlReader r)
{
  XElement x = (XElement) XNode.ReadFrom (r);
  ID = (int) x.Attribute ("id");
  FirstName = (string) x.Element ("firstname");
  LastName = (string) x.Element ("lastname");
}
```

XElement colabora con XmlReader para asegurarse de que los espacios de nombres se mantengan intactos y los prefijos se expandan correctamente, incluso si están definidos en un nivel externo. Entonces, si nuestro archivo XML se lee así:

```
<log xmlns="http://loggingspace">
  <logentry id="1">
  ...
```

Los XElements que construimos a nivel de logentry heredarían correctamente el espacio de nombres externo.

Uso de XmlWriter con XElement

Puedes usar un XElement simplemente para escribir elementos internos en un XmlWriter. El siguiente código escribe un millón de elementos logentry en un archivo XML utilizando XElement, sin almacenar todo en memoria:

```
using XmlWriter w = XmlWriter.Create ("logfile.xml");

w.WriteStartElement ("log");
for (int i = 0; i < 1000000; i++)
{
  XElement e = new XElement ("logentry",
                 new XAttribute ("id", i),
                 new XElement ("date", DateTime.Today.AddDays (-1)),
                 new XElement ("source", "test"));

  e.WriteTo (w);
}
w.WriteEndElement ();
```

El uso de XElement incurre en un mínimo costo adicional de ejecución. Si modificamos este ejemplo para usar XmlWriter en todo el proceso, no hay una diferencia mensurable en el tiempo de ejecución.

Trabajando con JSON

JSON se ha convertido en una alternativa popular a XML. Aunque carece de las características avanzadas de XML (como espacios de nombres, prefijos y esquemas), se beneficia de ser simple y claro, con un formato similar al que obtendrías al convertir un objeto JavaScript a una cadena.

Históricamente, .NET no tenía soporte incorporado para JSON, y tenías que depender de bibliotecas de terceros, principalmente Json.NET. Aunque esto ya no es cierto, la biblioteca Json.NET sigue siendo popular por varias razones:

- Ha estado disponible desde 2011.
- La misma API también funciona en versiones anteriores de .NET.
- Se considera más funcional (al menos en el pasado) que las API JSON de Microsoft.

Las API JSON de Microsoft tienen la ventaja de haber sido diseñadas desde cero para ser simples y extremadamente eficientes. Además, a partir de .NET 6, su funcionalidad se ha acercado mucho a la de Json.NET.

En esta sección, cubriremos lo siguiente:

- El lector y escritor de solo avance (Utf8JsonReader y Utf8JsonWriter)
- El lector de DOM de solo lectura JsonDocument
- El lector/escritor de DOM JsonNode

En "Serialización", en el suplemento en línea en http://www.albahari.com/nutshell, cubrimos JsonSerializer, que serializa y deserializa automáticamente JSON a clases.

Utf8JsonReader

System.Text.Json.Utf8JsonReader es un lector de solo avance optimizado para texto JSON codificado en UTF-8. Conceptualmente, es similar al XmlReader introducido anteriormente en este capítulo y se utiliza de manera muy similar.

Considere el siguiente archivo JSON llamado people.json:

```
{
  "FirstName":"Sara",
  "LastName":"Wells",
  "Age":35,
  "Friends":["Dylan","Ian"]
}
```

Las llaves indican un objeto JSON (que contiene propiedades como "FirstName" y "LastName"), mientras que los corchetes cuadrados indican un arreglo JSON (que contiene elementos repetidos). En este caso, los elementos repetidos son cadenas, pero podrían ser objetos (u otros arreglos).

El siguiente código analiza el archivo enumerando sus tokens JSON. Un token puede ser el inicio o fin de un objeto, el inicio o fin de un arreglo, el nombre de una propiedad o el valor de una propiedad o arreglo (cadena, número, true, false o null):

```
byte[] data = File.ReadAllBytes ("people.json");
Utf8JsonReader reader = new Utf8JsonReader (data);
while (reader.Read())
{
  switch (reader.TokenType)
  {
    case JsonTokenType.StartObject:
      Console.WriteLine ($"Start of object");
      break;
    case JsonTokenType.EndObject:
      Console.WriteLine ($"End of object");
      break;
    case JsonTokenType.StartArray:
      Console.WriteLine();
      Console.WriteLine ($"Start of array");
      break;
    case JsonTokenType.EndArray:
      Console.WriteLine ($"End of array");
      break;
    case JsonTokenType.PropertyName:
      Console.Write ($"Property: {reader.GetString()}");
      break;
    case JsonTokenType.String:
      Console.WriteLine ($" Value: {reader.GetString()}");
      break;
    case JsonTokenType.Number:
      Console.WriteLine ($" Value: {reader.GetInt32()}");
      break;
    default:
      Console.WriteLine ($"No support for {reader.TokenType}");
      break;
  }
}
```

Aquí está la salida:

```
Start of object
Property: FirstName Value: Sara
Property: LastName Value: Wells
Property: Age Value: 35
Property: Friends
Start of array
 Value: Dylan
 Value: Ian
End of array
End of object
```

Debido a que Utf8JsonReader trabaja directamente con UTF-8, recorre los tokens sin tener que convertir previamente la entrada a UTF-16 (el formato de las cadenas en .NET). La conversión a UTF-16 ocurre solo cuando se llama a un método como GetString().

Curiosamente, el constructor de Utf8JsonReader no acepta una matriz de bytes, sino un ReadOnlySpan<byte> (por esta razón, Utf8JsonReader se define como un struct ref). Puedes pasar una matriz de bytes porque hay una conversión implícita de T[] a ReadOnlySpan<T>. En el Capítulo 23, describimos cómo funcionan los spans y cómo se pueden utilizar para mejorar el rendimiento al minimizar las asignaciones de memoria.

JsonReaderOptions

De forma predeterminada, Utf8JsonReader requiere que el JSON cumpla estrictamente con el estándar JSON RFC 8259. Puedes indicarle al lector que sea más tolerante al pasar una instancia de JsonReaderOptions al constructor de Utf8JsonReader. Las opciones permiten lo siguiente:

Comentarios estilo C

> De forma predeterminada, los comentarios en JSON provocan que se lance una JsonException. Establecer la propiedad CommentHandling en JsonCommentHandling.Skip hace que los comentarios se ignoren, mientras que JsonCommentHandling.Allow hace que el lector los reconozca y emita tokens JsonTokenType.Comment cuando los encuentra. Los comentarios no pueden aparecer en medio de otros tokens.

Comas finales

> Según el estándar, la última propiedad de un objeto y el último elemento de un arreglo no deben tener una coma final. Establecer la propiedad AllowTrailingCommas en true relaja esta restricción.

Control sobre la profundidad máxima de anidamiento

> De forma predeterminada, los objetos y arreglos pueden anidarse hasta 64 niveles. Establecer MaxDepth a un número diferente anula esta configuración.

Utf8JsonWriter

System.Text.Json.Utf8JsonWriter es un escritor de JSON de solo avance. Admite los siguientes tipos de datos:

- String y DateTime (que se formatean como una cadena JSON)
- Los tipos numéricos Int32, UInt32, Int64, UInt64, Single, Double y Decimal (que se formatean como números JSON)
- bool (se formatea como literales JSON true/false)
- JSON null
- Arreglos

Puedes organizar estos tipos de datos en objetos de acuerdo con el estándar JSON. También te permite escribir comentarios, que no forman parte del estándar JSON pero que a menudo son compatibles con los analizadores de JSON en la práctica.

El siguiente código muestra cómo se utiliza:

```
var options = new JsonWriterOptions { Indented = true };

using (var stream = File.Create ("MyFile.json"))
using (var writer = new Utf8JsonWriter (stream, options))
{
  writer.WriteStartObject();
  // Property name and value specified in one call
  writer.WriteString ("FirstName", "Dylan");
  writer.WriteString ("LastName", "Lockwood");
  // Property name and value specified in separate calls
  writer.WritePropertyName ("Age");
  writer.WriteNumberValue (46);
  writer.WriteCommentValue ("This is a (non-standard) comment");
  writer.WriteEndObject();
}
```

Esto genera el siguiente archivo de salida:

```
{
  "FirstName": "Dylan",
  "LastName": "Lockwood",
  "Age": 46
  /*This is a (non-standard) comment*/
}
```

A partir de .NET 6, Utf8JsonWriter tiene un método llamado WriteRawValue para emitir directamente una cadena o una matriz de bytes en la secuencia JSON. Esto es útil en casos especiales, por ejemplo, si deseas que un número se escriba de manera que siempre incluya un punto decimal (1.0 en lugar de 1).

En este ejemplo, establecemos la propiedad Indented en JsonWriterOptions en true para mejorar la legibilidad. Si no lo hubiéramos hecho, la salida sería la siguiente:

```
{"FirstName":"Dylan","LastName":"Lockwood","Age":46...}
```

JsonWriterOptions también tiene una propiedad Encoder para controlar el escapado de las cadenas y una propiedad SkipValidation para permitir omitir las comprobaciones de validación estructural (lo que permite generar JSON de salida no válido).

JsonDocument

System.Text.Json.JsonDocument, por su parte, analiza los datos JSON en un DOM de solo lectura compuesto por instancias de JsonElement que se generan según la demanda. A diferencia de Utf8JsonReader, JsonDocument te permite acceder a los elementos de forma aleatoria.

JsonDocument es una de las dos APIs basadas en DOM para trabajar con JSON, siendo la otra JsonNode (que cubriremos en la siguiente sección). JsonNode se introdujo en .NET 6, principalmente para satisfacer la demanda de un DOM modificable. Sin embargo, también es adecuado para escenarios de solo lectura y expone una interfaz más fluida, respaldada por un DOM tradicional que utiliza clases para los valores JSON, las matrices y los objetos. En contraste, JsonDocument es extremadamente liviano, consta solo de una clase notable (JsonDocument) y dos estructuras livianas (JsonElement y JsonProperty) que analizan los datos subyacentes según sea necesario. La diferencia se ilustra en la Figura 11-1.



En la mayoría de los escenarios del mundo real, las ventajas de rendimiento de JsonDocument sobre JsonNode son insignificantes, por lo que puedes omitir JsonDocument y optar por JsonNode si prefieres aprender solo una API.

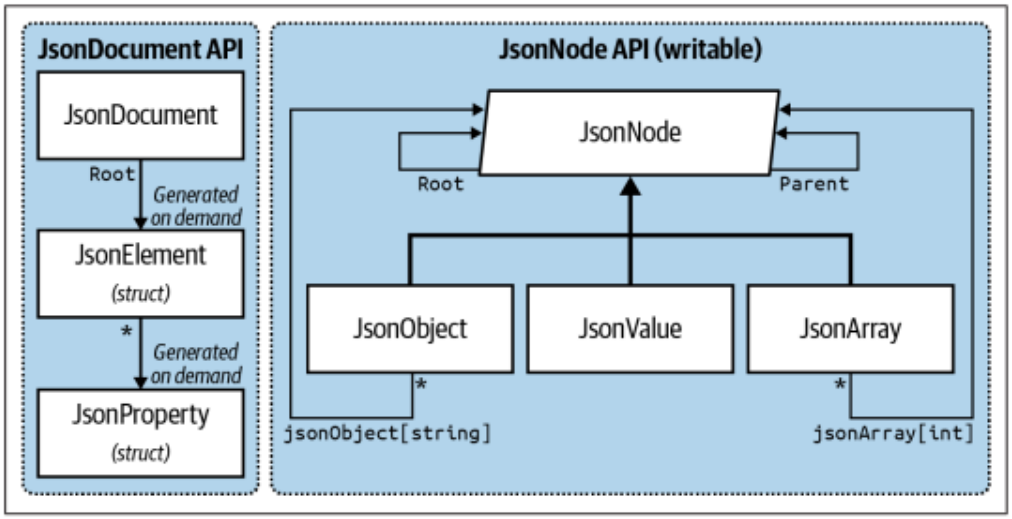


Figure 11-1. JSON DOM APIs



JsonDocument mejora aún más su eficiencia al emplear memoria en grupos para minimizar la recolección de basura. Esto significa que debes desechar el JsonDocument después de usarlo; de lo contrario, su memoria no se devolverá al grupo. En consecuencia, cuando una clase almacena un JsonDocument en un campo, también debe implementar IDisposable. Si esto resulta demasiado engorroso, considera utilizar JsonNode en su lugar.

El método estático `Parse` instancia un objeto `JsonDocument` a partir de un flujo (stream), una cadena (string) o un búfer de memoria:

```
using JsonDocument document = JsonDocument.Parse (jsonString);
...
```

Cuando llamas a `Parse`, opcionalmente puedes proporcionar un objeto `JsonDocumentOptions` para controlar el manejo de las comas finales, comentarios y la profundidad máxima de

anidamiento (para obtener más información sobre cómo funcionan estas opciones, consulta "JsonReaderOptions" en la página 546).

A partir de ahí, puedes acceder al DOM a través de la propiedad `RootElement`:

```
using JsonDocument document = JsonDocument.Parse ("123");
JsonElement root = document.RootElement;
Console.WriteLine (root.ValueKind);       // Number
```

`JsonElement` puede representar un valor JSON (cadena, número, verdadero/falso, nulo), un array u objeto; la propiedad `ValueKind` indica cuál es.



Los métodos que describiremos en las siguientes secciones lanzan una excepción si el elemento no es del tipo esperado. Si no estás seguro del esquema de un archivo JSON, puedes evitar tales excepciones comprobando primero `ValueKind` (o utilizando los métodos `TryGet*`).

`JsonElement` también proporciona dos métodos que funcionan para cualquier tipo de elemento: `GetRawText()` devuelve el JSON interno y `WriteTo` lo escribe en un `Utf8JsonWriter`.

Lectura de valores simples

Si el elemento representa un valor JSON, puedes obtener su valor llamando a los métodos `GetString`, `GetInt32`, `GetBoolean`, etc.:

```
using JsonDocument document = JsonDocument.Parse ("123");
int number = document.RootElement.GetInt32();
```

`JsonElement` también proporciona métodos para analizar cadenas JSON en otros tipos CLR comúnmente utilizados, como `DateTime` (e incluso binario en base64). También existen versiones `TryGet*` que evitan lanzar una excepción si el análisis falla.

Lectura de arrays JSON

Si el `JsonElement` representa un array, puedes llamar a los siguientes métodos:

EnumerateArray()

Enumera todos los subelementos de un array JSON (como `JsonElements`).

-GetArrayLength()

Devuelve el número de elementos en el array.

También puedes utilizar el indexador para devolver un elemento en una posición específica:

```
using JsonDocument document = JsonDocument.Parse (@"[1, 2, 3, 4, 5]");
int length = document.RootElement.GetArrayLength();   // 5
int value  = document.RootElement[3].GetInt32();      // 4
```

Lectura de objetos JSON

Si el `JsonElement` representa un objeto JSON, puedes llamar a los siguientes métodos:

EnumerateObject()

Enumera todos los nombres y valores de las propiedades del objeto.

GetProperty(string propertyName)

Obtiene una propiedad por su nombre (devuelve otro `JsonElement`). Lanza una excepción si el nombre no está presente.

TryGetProperty(string propertyName, out JsonElement value)

Devuelve la propiedad de un objeto si está presente.

Por ejemplo:

```
using JsonDocument document = JsonDocument.Parse (@"{ ""Age"": 32}");
JsonElement root = document.RootElement;
int age = root.GetProperty ("Age").GetInt32();
```

Aquí está cómo podríamos "descubrir" la propiedad "Age":

```
JsonProperty ageProp = root.EnumerateObject().First();
string name = ageProp.Name;             // Age
JsonElement value = ageProp.Value;
Console.WriteLine (value.ValueKind);    // Number
Console.WriteLine (value.GetInt32());   // 32
```

JsonDocument y LINQ

`JsonDocument` se presta bien para su uso con LINQ. Dado el siguiente archivo JSON:

```
[
  {
    "FirstName":"Sara",
    "LastName":"Wells",
    "Age":35,
    "Friends":["Ian"]
  },
  {
    "FirstName":"Ian",
    "LastName":"Weems",
    "Age":42,
    "Friends":["Joe","Eric","Li"]
  },
  {
    "FirstName":"Dylan",
    "LastName":"Lockwood",
    "Age":46,
    "Friends":["Sara","Ian"]


  }
]
```

Podemos utilizar `JsonDocument` para realizar consultas con LINQ de la siguiente manera:

```
using var stream = File.OpenRead (jsonPath);
using JsonDocument document = JsonDocument.Parse (json);

var query =
  from person in document.RootElement.EnumerateArray()
  select new
  {
    FirstName = person.GetProperty ("FirstName").GetString(),
    Age = person.GetProperty ("Age").GetInt32(),
    Friends =
      from friend in person.GetProperty ("Friends").EnumerateArray()
      select friend.GetString()
  };
```

Dado que las consultas de LINQ se evalúan de forma diferida (lazy evaluation), es importante enumerar la consulta antes de que el documento quede fuera de alcance y `JsonDocument` se elimine implícitamente mediante la declaración `using`.

Realizando actualizaciones con un escritor de JSON

Aunque `JsonDocument` es de solo lectura, puedes enviar el contenido de un `JsonElement` a un `Utf8JsonWriter` utilizando el método `WriteTo`. Esto proporciona un mecanismo para emitir una versión modificada del JSON. A continuación se muestra cómo podemos tomar el JSON del ejemplo anterior y escribirlo en un nuevo archivo JSON que incluye solo personas con dos o más amigos:

```
using var json = File.OpenRead (jsonPath);
using JsonDocument document = JsonDocument.Parse (json);

var options = new JsonWriterOptions { Indented = true };

using (var outputStream = File.Create ("NewFile.json"))
using (var writer = new Utf8JsonWriter (outputStream, options))
{
  writer.WriteStartArray();
  foreach (var person in document.RootElement.EnumerateArray())
  {
    int friendCount = person.GetProperty ("Friends").GetArrayLength();
    if (friendCount >= 2)
      person.WriteTo (writer);
  }
}
```

Sin embargo, si necesitas la capacidad de actualizar el DOM, `JsonNode` es una mejor solución.

JsonNode

`JsonNode` (en `System.Text.Json.Nodes`) fue introducido en .NET 6, principalmente para satisfacer la demanda de un DOM editable. Sin embargo, también es adecuado en escenarios de solo lectura. JsonNode proporciona una interfaz más fluida respaldada por un DOM tradicional que utiliza clases para representar los valores, arrays y objetos JSON (ver Figura 11-1). Al ser clases, incurren en un costo de recolección de basura, pero esto probablemente sea insignificante en la mayoría de los escenarios del mundo real. JsonNode sigue siendo altamente optimizado y puede ser más rápido que JsonDocument cuando los mismos nodos se leen repetidamente (porque JsonNode, aunque perezoso, almacena en caché los resultados del análisis).

El método estático Parse crea un JsonNode a partir de un stream, cadena de texto, búfer de memoria o Utf8JsonReader:

```
JsonNode node = JsonNode.Parse (jsonString);
```

Al llamar a Parse, opcionalmente puedes proporcionar un objeto JsonDocumentOptions para controlar el manejo de las comas finales, los comentarios y la profundidad máxima de anidamiento (para más información sobre cómo funcionan estas opciones, consulta "JsonReaderOptions" en la página 546). A diferencia de JsonDocument, JsonNode no requiere ser liberado (disposable).



Al llamar a ToString() en un JsonNode se obtiene una cadena JSON legible para humanos (con formato). También existe un método ToJsonString() que devuelve una cadena JSON compacta.

Parse devuelve un subtipo de JsonNode, que puede ser JsonValue, JsonObject o JsonArray. Para evitar tener que hacer un cast, JsonNode proporciona métodos de ayuda llamados AsValue(), AsObject() y AsArray():

```
var node = JsonNode.Parse ("123");  // Parses to a JsonValue
int number = node.AsValue().GetValue<int>();
// Shortcut for ((JsonValue)node).GetValue<int>();
```

Sin embargo, por lo general no es necesario llamar a estos métodos, ya que los miembros más comúnmente utilizados se exponen directamente en la clase JsonNode:

```
var node = JsonNode.Parse ("123");
int number = node.GetValue<int>();
// Shortcut for node.AsValue().GetValue<int>();
```

Leyendo valores simples

Acabamos de ver que puedes extraer o analizar un valor simple llamando a GetValue con un parámetro de tipo. Para hacer esto aún más fácil, JsonNode sobrecarga los operadores de conversión explícita de C#, lo que permite el siguiente atajo:

```
var node = JsonNode.Parse ("123");
int number = (int) node;
```

Los tipos para los cuales esto funciona incluyen los tipos numéricos estándar: char, bool, DateTime, DateTimeOffset y Guid (y sus versiones anulables), así como string.

Si no estás seguro de si el análisis tendrá éxito, se requiere el siguiente código:

```
if (node.AsValue().TryGetValue<int> (out var number))
  Console.WriteLine (number);
```



Los nodos que se han analizado a partir de un texto JSON están respaldados internamente por un JsonElement (parte de la API de solo lectura JsonDocument). Puedes extraer el JsonElement subyacente de la siguiente manera:

```
JsonElement je = node.GetValue<JsonElement>();
```

Sin embargo, esto no funciona cuando el nodo se instancia explícitamente (como será el caso cuando actualicemos el DOM). Estos nodos no están respaldados por un JsonElement, sino por el valor analizado real (ver "Realizar actualizaciones con JsonNode" en la página 554).

Leyendo arrays JSON

Un JsonNode que representa un array JSON será de tipo JsonArray. JsonArray implementa IList<JsonNode>, por lo que puedes enumerarlo y acceder a los elementos como si fuera un array o una lista:

```
var node = JsonNode.Parse (@"[1, 2, 3, 4, 5]");
Console.WriteLine (node.AsArray().Count);       // 5

foreach (JsonNode child in node.AsArray())
{ ... }
```

Como atajo, puedes acceder al índice directamente desde la clase JsonNode:

```
Console.WriteLine ((int)node[0]);   // 1
```

Leyendo objetos JSON

Un JsonNode que representa un objeto JSON será de tipo JsonObject. JsonObject implementa IDictionary<string, JsonNode>, por lo que puedes acceder a un miembro a través del índice y enumerar los pares clave/valor del diccionario.

Y al igual que con JsonArray, puedes acceder al índice directamente desde la clase JsonNode:

```
var node = JsonNode.Parse (@"{ ""Name"":""Alice"", ""Age"": 32}");
string name = (string) node ["Name"];   // Alice
int age = (int) node ["Age"];           // 32
```

Así es como podríamos "descubrir" las propiedades Name y Age:

```
// Enumerate over the dictionary's key/value pairs:
foreach (KeyValuePair<string,JsonNode> keyValuePair in node.AsObject())
{
  string propertyName = keyValuePair.Key;   // "Name" (then "Age")
  JsonNode value = keyValuePair.Value;
}
```

Si no estás seguro de si una propiedad ha sido definida, también puedes utilizar el siguiente patrón:

```
if (node.AsObject().TryGetPropertyValue ("Name", out JsonNode nameNode))
{ ... }
```

Fluent traversal and LINQ

Puedes navegar profundamente en una jerarquía utilizando solo indexadores. Por ejemplo, dado el siguiente archivo JSON:

```
[
  {
    "FirstName":"Sara",
    "LastName":"Wells",
    "Age":35,
    "Friends":["Ian"]
  },
  {
    "FirstName":"Ian",
    "LastName":"Weems",
    "Age":42,
    "Friends":["Joe","Eric","Li"]
  },
  {
    "FirstName":"Dylan",
    "LastName":"Lockwood",
    "Age":46,
    "Friends":["Sara","Ian"]
  }
]
```

Podemos extraer el tercer amigo de la segunda persona de la siguiente manera:

```
string li = (string) node[1]["Friends"][2];
```

Un archivo como este también es fácil de consultar mediante LINQ:

```
JsonNode node = JsonNode.Parse (File.ReadAllText (jsonPath));

var query =
  from person in node.AsArray()
  select new
  {
    FirstName = (string) person ["FirstName"],
    Age = (int) person ["Age"],
    Friends =
      from friend in person ["Friends"].AsArray()
      select (string) friend
  };
```

A diferencia de JsonDocument, JsonNode no es desechable (no implementa IDisposable), por lo que no tenemos que preocuparnos por la eliminación potencial durante la enumeración perezosa.

Realizando actualizaciones con JsonNode

JsonObject y JsonArray son mutables, por lo que puedes actualizar su contenido.

La forma más sencilla de reemplazar o agregar propiedades a un JsonObject es a través del indexador. En el siguiente ejemplo, cambiamos el valor de la propiedad "Color" de "Red" a "White" y agregamos una nueva propiedad llamada "Valid":

```
var node = JsonNode.Parse ("{ \"Color\": \"Red\" }");
node ["Color"] = "White";
node ["Valid"] = true;
Console.WriteLine (node.ToJsonString());  // {"Color":"White","Valid":true}
```

La segunda línea de ese ejemplo es un atajo para lo siguiente:

```
node ["Color"] = JsonValue.Create ("White");
```

En lugar de asignar un valor simple a la propiedad, también puedes asignarle un JsonArray o JsonObject. (Demostraremos cómo construir instancias de JsonArray y JsonObject en la siguiente sección).

Para eliminar una propiedad, primero debes convertirla a JsonObject (o llamar a AsObject) y luego llamar al método Remove:

```
node.AsObject().Remove ("Valid");
```

(JsonObject también expone un método Add, que arroja una excepción si la propiedad ya existe).

JsonArray también te permite usar el indexador para reemplazar elementos:

```
var node = JsonNode.Parse ("[1, 2, 3]");
node[0] = 10;
```

Al llamar a AsArray, se exponen los métodos Add/Insert/Remove/RemoveAt. En el siguiente ejemplo, eliminamos el primer elemento del array y agregamos uno al final:

```
var arrayNode = JsonNode.Parse ("[1, 2, 3]");
arrayNode.AsArray().RemoveAt(0);
arrayNode.AsArray().Add (4);
Console.WriteLine (arrayNode.ToJsonString());  // [2,3,4]
```

Construyendo un JsonNode DOM de forma programática

JsonArray y JsonObject tienen constructores que admiten la sintaxis de inicialización de objetos, lo cual te permite construir todo el DOM de JsonNode en una sola expresión:

```
var node = new JsonArray
{
  new JsonObject {
    ["Name"] = "Tracy",
    ["Age"] = 30,
    ["Friends"] = new JsonArray ("Lisa", "Joe")
  },
  new JsonObject {
    ["Name"] = "Jordyn",
    ["Age"] = 25,
    ["Friends"] = new JsonArray ("Tracy", "Li")
  }
};
```

Esto se evalúa en el siguiente JSON:

```
[
  {
    "Name": "Tracy",
    "Age": 30,
    "Friends": ["Lisa", "Joe"]
  },
  {
    "Name": "Jordyn",
    "Age": 25,
    "Friends": ["Tracy","Li"]
  }
]
```

12 Eliminación y recolección de basura (Disposal and Garbage Collection)

Algunos objetos requieren código explícito de liberación de recursos, como archivos abiertos, bloqueos, manipuladores del sistema operativo y objetos no administrados. En el lenguaje .NET, esto se conoce como eliminación y se admite a través de la interfaz IDisposable. La memoria administrada ocupada por objetos no utilizados también debe ser recuperada en algún momento; esta función se conoce como recolección de basura y es realizada por el CLR.

La eliminación difiere de la recolección de basura en que la eliminación generalmente se inicia de forma explícita, mientras que la recolección de basura es totalmente automática. En otras palabras, el programador se encarga de liberar recursos como manipuladores de archivos, bloqueos y recursos del sistema operativo, mientras que el CLR se encarga de liberar memoria.

Este capítulo discute tanto la eliminación como la recolección de basura, y también describe los finalizadores de C# y el patrón mediante el cual pueden proporcionar una opción de respaldo para la eliminación. Por último, se analizan las complejidades del recolector de basura y otras opciones de administración de memoria.

IDisposable, Dispose y Close

.NET define una interfaz especial para tipos que requieren un método de eliminación:

```
public interface IDisposable
{
  void Dispose();
}
```

La instrucción using de C# proporciona un atajo sintáctico para llamar a Dispose en objetos que implementan IDisposable, utilizando un bloque try/finally:

```
using (FileStream fs = new FileStream ("myFile.txt", FileMode.Open))
{
  // ... Write to the file ...
}
```

El compilador convierte esto a lo siguiente:

```
FileStream fs = new FileStream ("myFile.txt", FileMode.Open);
try
{
  // ... Write to the file ...
}
finally
{
  if (fs != null) ((IDisposable)fs).Dispose();
}
```

El bloque finally asegura que el método Dispose sea llamado incluso cuando se lance una excepción o el código salga del bloque de forma anticipada.

De manera similar, la siguiente sintaxis garantiza la eliminación tan pronto como fs queda fuera del ámbito:

```
using FileStream fs = new FileStream ("myFile.txt", FileMode.Open);

// ... Write to the file ...
```

En escenarios simples, escribir tu propio tipo desechable es simplemente cuestión de implementar IDisposable y escribir el método Dispose:

```
sealed class Demo : IDisposable
{
  public void Dispose()
  {
    // Perform cleanup / tear-down.
    ...
  }
}
```



Este patrón funciona bien en casos simples y es apropiado para clases selladas. En "Llamando a Dispose desde un finalizador" en la página 566, describimos un patrón más elaborado que puede proporcionar una opción de respaldo para los consumidores que olvidan llamar a Dispose. Con tipos no sellados, hay un fuerte argumento para seguir este último patrón desde el principio; de lo contrario, se vuelve muy complicado si el subtipo desea agregar dicha funcionalidad por sí mismo.

Semántica estándar de eliminación

.NET sigue un conjunto de reglas de facto en su lógica de eliminación. Estas reglas no están cableadas en .NET ni en el lenguaje C# de ninguna manera; su propósito es definir un protocolo consistente para los consumidores. Aquí están:

1. Después de que un objeto ha sido eliminado, está más allá de la redención. No se puede reactivar y llamar a sus métodos o propiedades (excepto Dispose) arroja una ObjectDisposedException.

2. Llamar al método Dispose de un objeto repetidamente no causa ningún error.

3. Si el objeto desechable x "posee" al objeto desechable y, el método Dispose de x llama automáticamente al método Dispose de y, a menos que se indique lo contrario.

Estas reglas también son útiles al escribir tus propios tipos, aunque no son obligatorias. Nada te impide escribir un método "Undispose" excepto, tal vez, las críticas que puedas recibir de tus colegas.

Según la regla 3, un objeto contenedor automáticamente elimina sus objetos secundarios. Un buen ejemplo es un control contenedor de Windows Forms, como un formulario o un panel. El contenedor puede alojar muchos controles secundarios, pero no necesitas eliminar explícitamente cada uno de ellos; cerrar o eliminar el control o formulario principal se encarga de todo. Otro ejemplo es cuando envuelves un FileStream en un DeflateStream. Al eliminar el

DeflateStream, también se elimina el FileStream, a menos que se indique lo contrario en el constructor.

Cerrar y Detener

Algunos tipos definen un método llamado "Close" además de "Dispose". La biblioteca de clases de .NET no es completamente consistente en cuanto a la semántica de un método "Close", aunque en casi todos los casos es uno de los siguientes:

- Funcionalmente idéntico a "Dispose".
- Un subconjunto funcional de "Dispose".

Un ejemplo de esto último es IDbConnection: una conexión cerrada puede volver a abrirse, pero una conexión eliminada no puede. Otro ejemplo es un formulario de Windows que se activa con "ShowDialog": "Close" lo oculta y "Dispose" libera sus recursos.

Algunas clases definen un método "Stop" (por ejemplo, Timer o HttpListener). Un método "Stop" puede liberar recursos no administrados, al igual que "Dispose", pero a diferencia de "Dispose", permite reiniciar.

Cuándo utilizar Dispose

Una regla segura a seguir (en casi todos los casos) es "si tienes dudas, elimina". Los objetos que envuelven un recurso no administrado casi siempre requerirán eliminación para liberar el recurso. Ejemplos incluyen flujos de archivos o de red, sockets de red, controles de Windows Forms, pinceles y mapas de bits de GDI+. Por otro lado, si un tipo es desechable, a menudo (pero no siempre) hará referencia a un identificador no administrado, directa o indirectamente. Esto se debe a que los identificadores no administrados proporcionan el acceso al "mundo exterior" de los recursos del sistema operativo, conexiones de red y bloqueos de bases de datos; los principales medios por los cuales los objetos pueden causar problemas fuera de sí mismos si se abandonan incorrectamente.

Sin embargo, hay tres escenarios en los que no se debe realizar la eliminación:

- Cuando no "posees" el objeto, por ejemplo, al obtener un objeto compartido a través de un campo o propiedad estática.
- Cuando el método Dispose de un objeto hace algo que no deseas.
- Cuando el método Dispose de un objeto es innecesario por diseño y eliminar ese objeto añadiría complejidad a tu programa.

La primera categoría es rara. Los casos principales se encuentran en el espacio de nombres System.Drawing: los objetos de GDI+ obtenidos a través de campos o propiedades estáticas (como Brushes.Blue) nunca deben eliminarse porque se utiliza la misma instancia durante toda la vida de la aplicación. Sin embargo, las instancias que obtienes a través de constructores (como new SolidBrush) deben eliminarse, al igual que las instancias obtenidas a través de métodos estáticos (como Font.FromHdc).

La segunda categoría es más común. Hay algunos buenos ejemplos en los espacios de nombres System.IO y System.Data:

Type	Disposal function	When not to dispose
`MemoryStream`	Prevents further I/O	When you later need to read/write the stream
`StreamReader`, `StreamWriter`	Flushes the reader/writer and closes the underlying stream	When you want to keep the underlying stream open (you must then call `Flush` on a `StreamWriter` when you're done)
`IDbConnection`	Releases a database connection and clears the connection string	If you need to re-Open it, you should call `Close` instead of `Dispose`
`DbContext` (EF Core)	Prevents further use	When you might have lazily evaluated queries connected to that context

El método Dispose de MemoryStream solo deshabilita el objeto; no realiza ninguna limpieza crítica porque un MemoryStream no tiene identificadores no administrados u otros recursos similares.

La tercera categoría incluye las clases como StringReader y StringWriter. Estos tipos son desechables debido a la clase base en lugar de una necesidad genuina de realizar una limpieza esencial. Si instancias y trabajas con dicho objeto completamente en un método, envolverlo en un bloque "using" agrega poca inconveniencia. Pero si el objeto tiene una duración más larga, realizar un seguimiento de cuándo ya no se utiliza para poder eliminarlo agrega complejidad innecesaria. En esos casos, simplemente puedes ignorar la eliminación del objeto.



Ignorar la eliminación a veces puede tener un costo de rendimiento

(consultar "Llamar a Dispose desde un finalizador" en la página 566).

Limpiar campos en la eliminación

En general, no es necesario limpiar los campos de un objeto en su método Dispose. Sin embargo, es una buena práctica cancelar la suscripción a eventos a los que el objeto se ha suscrito internamente a lo largo de su vida útil (para un ejemplo, consultar "Fugas de memoria administrada" en la página 576). Al cancelar la suscripción a dichos eventos, se evita recibir notificaciones de eventos no deseadas y se evita mantener accidentalmente el objeto con vida a ojos del recolector de basura (GC).



Un método Dispose en sí mismo no libera la memoria (administrada);

esto solo ocurre en la recolección de basura.

También es útil establecer un campo para indicar que el objeto se ha eliminado, de modo que puedas lanzar una ObjectDisposedException si un consumidor intenta llamar a miembros del objeto más adelante. Un buen patrón es utilizar una propiedad automática de solo lectura para esto:

```
public bool IsDisposed { get; private set; }
```

Aunque técnicamente innecesario, también puede ser bueno limpiar los controladores de eventos propios del objeto (estableciéndolos en null) en el método Dispose. Esto elimina la posibilidad de que esos eventos se activen durante o después de la eliminación.

Ocasionalmente, un objeto puede contener secretos de alto valor, como claves de encriptación. En estos casos, tiene sentido eliminar dichos datos de los campos durante la eliminación (para evitar su posible descubrimiento por otros procesos en la máquina cuando la memoria se libera posteriormente al sistema operativo). La clase SymmetricAlgorithm en System.Security.Cryptography hace exactamente esto al llamar a Array.Clear en la matriz de bytes que contiene la clave de encriptación.

Eliminación anónima

A veces, es útil implementar IDisposable sin tener que escribir una clase. Por ejemplo, supongamos que deseas exponer métodos en una clase que suspendan y reanuden el procesamiento de eventos:

```
class Foo
{
  int _suspendCount;

  public void SuspendEvents() => _suspendCount++;
  public void ResumeEvents() => _suspendCount--;

  void FireSomeEvent()
  {
    if (_suspendCount == 0)
      ... fire some event ...
  }
  ...
}
```

Una API así es incómoda de usar. Los consumidores deben recordar llamar a ResumeEvents. Y para ser robustos, deben hacerlo dentro de un bloque "finally" (en caso de que se lance una excepción):

```
var foo = new Foo();
foo.SuspendEvents();
try
{

  ... do stuff ...      // Because an exception could be thrown here
}
finally
{
  foo.ResumeEvents();   // ...we must call this in a finally block
}
```

Un mejor patrón es prescindir de ResumeEvents y hacer que SuspendEvents devuelva un IDisposable. Los consumidores pueden hacer lo siguiente:

```
using (foo.SuspendEvents())
{
  ... do stuff ...
}
```

El problema es que esto traslada trabajo a quien implementa el método SuspendEvents. Incluso con un esfuerzo por reducir el espacio en blanco, terminamos con este desorden adicional:

```
public IDisposable SuspendEvents()
{
  _suspendCount++;
  return new SuspendToken (this);
}

class SuspendToken : IDisposable
{
  Foo _foo;
  public SuspendToken (Foo foo) => _foo = foo;
  public void Dispose()
  {
    if (_foo != null) _foo._suspendCount--;
    _foo = null;  // Prevent against consumer disposing twice
  }
}
```

El patrón de eliminación anónima resuelve este problema. Con la siguiente clase reutilizable:

```
public class Disposable : IDisposable
{
  public static Disposable Create (Action onDispose)
    => new Disposable (onDispose);

  Action _onDispose;
  Disposable (Action onDispose) => _onDispose = onDispose;

  public void Dispose()
  {
    _onDispose?.Invoke();   // Execute disposal action if non-null.
    _onDispose = null;      // Ensure it can't execute a second time.
  }
}
```

podemos reducir nuestro método SuspendEvents a lo siguiente:

```
public IDisposable SuspendEvents()
{
  _suspendCount++;
  return Disposable.Create (() => _suspendCount--);
}
```

Recolección automática de basura

Independientemente de si un objeto requiere un método Dispose para la lógica personalizada de limpieza, en algún momento la memoria que ocupa en el montón debe ser liberada. El CLR se encarga de esto completamente de forma automática mediante una recolección automática de basura. Nunca desasignas memoria administrada tú mismo. Por ejemplo, considera el siguiente método:

```
public void Test()
{
  byte[] myArray = new byte[1000];
  ...
}
```

Cuando se ejecuta Test, se asigna un arreglo para contener 1.000 bytes en el montón de memoria. El arreglo está referenciado por la variable `myArray`, almacenada en la pila de variables locales. Cuando el método sale, esta variable local `myArray` sale de alcance, lo que significa que ya no queda nada para referenciar el arreglo en el montón de memoria. El arreglo huérfano se vuelve elegible para ser reclamado en la recolección de basura.



En el modo de depuración con las optimizaciones deshabilitadas, la duración de un objeto referenciado por una variable local se extiende hasta el final del bloque de código para facilitar la depuración. De lo contrario, se vuelve elegible para la recolección en el punto más temprano en el que ya no se utiliza.

La recolección de basura no ocurre inmediatamente después de que un objeto queda huérfano. Al igual que la recolección de basura en la calle, ocurre periódicamente, aunque (a diferencia de la recolección de basura en la calle) no en un horario fijo. El CLR basa su decisión sobre cuándo realizar la recolección en varios factores, como la memoria disponible, la cantidad de asignación de memoria y el tiempo transcurrido desde la última recolección (el GC se ajusta automáticamente para optimizar los patrones de acceso a la memoria específicos de una aplicación). Esto significa que hay un retraso indeterminado entre el momento en que un objeto queda huérfano y su liberación de memoria. Este retraso puede variar desde nanosegundos hasta días.



El GC no recolecta toda la basura en cada recolección. En su lugar, el administrador de memoria divide los objetos en generaciones, y el GC recolecta las generaciones nuevas (objetos asignados recientemente) con mayor frecuencia que las generaciones antiguas (objetos de larga duración). Discutiremos esto con más detalle en "Cómo funciona el GC" en la página 569.

Recolección de basura y consumo de memoria

El GC intenta encontrar un equilibrio entre el tiempo que dedica a realizar la recolección de basura y el consumo de memoria de la aplicación (conjunto de trabajo). En consecuencia, las aplicaciones pueden consumir más memoria de la necesaria, especialmente si se construyen matrices temporales grandes.

Puedes monitorizar el consumo de memoria de un proceso mediante el Administrador de tareas de Windows o el Monitor de recursos, o programáticamente consultando un contador de rendimiento:

```
// These types are in System.Diagnostics:
string procName = Process.GetCurrentProcess().ProcessName;
using PerformanceCounter pc = new PerformanceCounter
      ("Process", "Private Bytes", procName);
Console.WriteLine (pc.NextValue());
```

Esto consulta el conjunto de trabajo privado, que proporciona la mejor indicación general del consumo de memoria de tu programa. Específicamente, excluye la memoria que el CLR ha desasignado internamente y está dispuesto a devolver al sistema operativo si otro proceso la necesita.

Raíces (Roots)

Una raíz es algo que mantiene vivo a un objeto. Si un objeto no es referenciado directa o indirectamente por una raíz, será elegible para la recolección de basura.

Una raíz puede ser uno de los siguientes:

- Una variable local o parámetro en un método en ejecución (o en cualquier método de su pila de llamadas)
- Una variable estática
- Un objeto en la cola que almacena objetos listos para finalización (ver la siguiente sección)

Es imposible que el código se ejecute en un objeto eliminado, por lo que si existe alguna posibilidad de que un método (de instancia) se ejecute, su objeto debe estar de alguna manera referenciado de una de estas formas.

Ten en cuenta que un grupo de objetos que se referencian entre sí de forma cíclica se consideran muertos sin una referencia raíz (ver Figura 12-1). Dicho de otra manera, los objetos a los que no se puede acceder siguiendo las flechas (referencias) desde un objeto raíz son inalcanzables y, por lo tanto, susceptibles de ser recolectados.

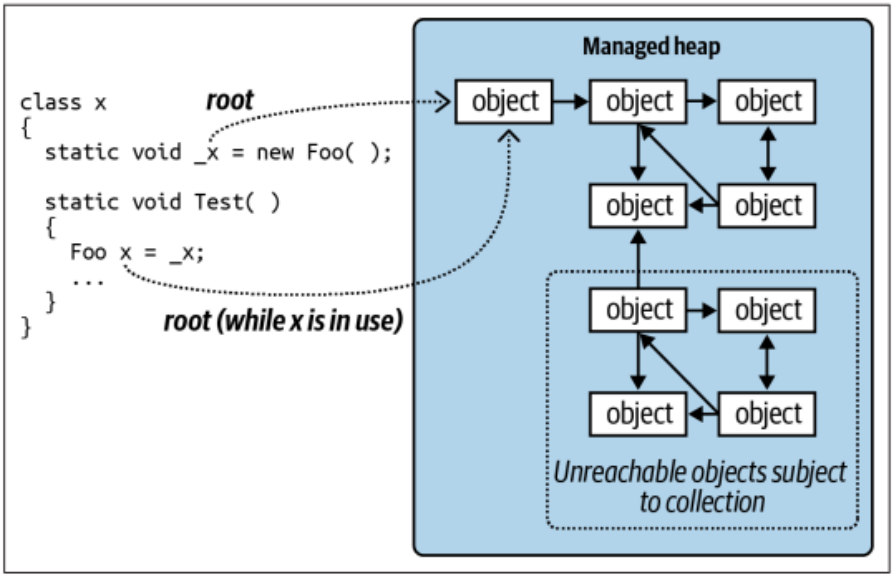


Figure 12-1. Roots

Finalizadores (Finalizers)

Antes de que un objeto sea liberado de la memoria, se ejecuta su finalizador, si tiene uno. Un finalizador se declara de manera similar a un constructor, pero se le antepone el símbolo ~:

```
class Test
{
  ~Test()
  {
    // Finalizer logic...
  }
}
```

(Aunque la declaración de un finalizador es similar a la de un constructor, los finalizadores no pueden ser declarados como públicos o estáticos, no pueden tener parámetros y no pueden llamar a la clase base).

Los finalizadores son posibles porque la recolección de basura funciona en fases distintas. Primero, el GC identifica los objetos no utilizados listos para ser eliminados. Aquellos sin finalizadores se eliminan inmediatamente. Aquellos con finalizadores pendientes (no ejecutados) se mantienen vivos (por ahora) y se colocan en una cola especial.

En ese momento, la recolección de basura se completa y tu programa continúa ejecutándose. Luego, el subproceso del finalizador entra en acción y comienza a ejecutarse en paralelo a tu programa, tomando objetos de esa cola especial y ejecutando sus métodos de finalización.

Antes de que se ejecute el finalizador de cada objeto, este todavía está completamente vivo, ya que esa cola actúa como un objeto raíz. Después de haber sido retirado de la cola y ejecutado el finalizador, el objeto se convierte en un objeto huérfano y será eliminado en la próxima recolección (para la generación de ese objeto).

Los finalizadores pueden ser útiles, pero vienen con algunas advertencias:

- Los finalizadores ralentizan la asignación y recolección de memoria (el GC necesita realizar un seguimiento de qué finalizadores se han ejecutado).
- Los finalizadores prolongan la vida del objeto y cualquier objeto al que haga referencia (todos deben esperar al siguiente "camión de basura" para ser eliminados realmente).
- Es imposible predecir en qué orden se llamarán los finalizadores de un conjunto de objetos.
- Tienes un control limitado sobre cuándo se llamará el finalizador de un objeto.
- Si el código en un finalizador bloquea, otros objetos no podrán finalizarse.
- Los finalizadores pueden evitarse por completo si una aplicación no se descarga correctamente.

En resumen, los finalizadores son algo parecido a los abogados: aunque hay casos en los que realmente los necesitas, en general no quieres usarlos a menos que sea absolutamente necesario. Si los usas, debes estar seguro al 100% de entender lo que están haciendo por ti.

Aquí tienes algunas pautas para implementar finalizadores:

- Asegúrate de que tu finalizador se ejecute rápidamente.
- Nunca bloquees en tu finalizador (ver "Bloqueo" en la página 612).
- No hagas referencia a otros objetos finalizables.

- No lances excepciones.


El CLR puede llamar al finalizador de un objeto incluso si se produce una excepción durante la construcción. Por esta razón, es mejor no asumir que los campos están correctamente inicializados al escribir un finalizador.

Llamar a Dispose desde un finalizador (Calling Dispose from a Finalizer)

Llamar a Dispose desde un finalizador es un patrón común. Esto tiene sentido cuando la limpieza no es urgente y acelerarla llamando a Dispose es más una optimización que una necesidad.


Ten en cuenta que con este patrón estás vinculando la liberación de memoria con la liberación de recursos, dos cosas con intereses potencialmente divergentes (a menos que el recurso sea memoria en sí). También aumentas la carga en el hilo de finalización.

Este patrón también sirve como respaldo para los casos en los que un consumidor simplemente olvida llamar a Dispose. Sin embargo, es una buena idea registrar el error para poder corregir el error.

Hay un patrón estándar para implementar esto, que es el siguiente:

```
class Test : IDisposable
{
  public void Dispose()             // NOT virtual
  {
    Dispose (true);
    GC.SuppressFinalize (this);     // Prevent finalizer from running.
  }

  protected virtual void Dispose (bool disposing)
  {
    if (disposing)
    {
      // Call Dispose() on other objects owned by this instance.
      // You can reference other finalizable objects here.
      // ...
    }

    // Release unmanaged resources owned by (just) this object.
    // ...
  }

  ~Test() => Dispose (false);
}
```

La clase Dispose tiene una sobrecarga que acepta un indicador bool llamado "disposing". La versión sin parámetros no se declara como virtual y simplemente llama a la versión mejorada con el valor "true". La versión mejorada contiene la lógica real de liberación de recursos y es protegida y virtual, lo cual proporciona un punto seguro para que las subclases agreguen su

propia lógica de liberación de recursos. El indicador "disposing" significa que se está llamando "correctamente" desde el método Dispose en lugar de en el "modo de último recurso" desde el finalizador. La idea es que cuando se llame con disposing establecido en "false", este método en general no debería hacer referencia a otros objetos con finalizadores (porque dichos objetos podrían haber sido finalizados y estar en un estado impredecible). Esto descarta bastante. Aquí hay un par de tareas que el método Dispose aún puede realizar en el modo de último recurso, cuando disposing es "false":

- Liberar cualquier referencia directa a recursos del sistema operativo (obtenidos, quizás, mediante una llamada P/Invoke a la API de Win32).
- Eliminar un archivo temporal creado durante la construcción.

Para que esto sea robusto, cualquier código que pueda generar una excepción debe estar envuelto en un bloque try/catch, y la excepción idealmente debería registrarse en un registro. Cualquier registro debe ser lo más simple y robusto posible.

Observa que llamamos a GC.SuppressFinalize en el método Dispose sin parámetros, esto evita que el finalizador se ejecute cuando el recolector de basura lo alcance más adelante. Técnicamente, esto no es necesario dado que los métodos Dispose deben tolerar llamadas repetidas. Sin embargo, hacerlo mejora el rendimiento porque permite que el objeto (y los objetos a los que hace referencia) sean recolectados en un solo ciclo.

Resurrección

Supongamos que un finalizador modifica un objeto vivo de manera que hace referencia de nuevo al objeto moribundo. Cuando ocurre la siguiente recolección de basura (para la generación del objeto), el CLR verá que el objeto previamente moribundo ya no está huérfano, por lo que evitará la recolección de basura. Este es un escenario avanzado y se conoce como resurrección.

Para ilustrar esto, supongamos que queremos escribir una clase que gestione un archivo temporal. Cuando una instancia de esa clase sea recolectada por el recolector de basura, nos gustaría que el finalizador elimine el archivo temporal. Suena fácil:

```
public class TempFileRef
{
  public readonly string FilePath;
  public TempFileRef (string filePath) { FilePath = filePath; }

  ~TempFileRef() { File.Delete (FilePath); }
}
```

Desafortunadamente, esto tiene un error: `File.Delete` podría lanzar una excepción (debido a falta de permisos, tal vez, o porque el archivo está en uso o ya ha sido eliminado). Tal excepción podría hacer que la aplicación completa falle (además de evitar que se ejecuten otros finalizadores). Podríamos simplemente "ignorar" la excepción con un bloque `catch` vacío, pero entonces nunca sabríamos que algo salió mal. Llamar a alguna API elaborada de informes de errores también sería indeseable, ya que sobrecargaría el hilo del finalizador, obstaculizando la recolección de basura para otros objetos. Queremos restringir las acciones de finalización a aquellas que sean simples, confiables y rápidas.

Una mejor opción es registrar el fallo en una colección estática, como se muestra a continuación:

```
public class TempFileRef
{
  static internal readonly ConcurrentQueue<TempFileRef> FailedDeletions
    = new ConcurrentQueue<TempFileRef>();

  public readonly string FilePath;
  public Exception DeletionError { get; private set; }

  public TempFileRef (string filePath) { FilePath = filePath; }

  ~TempFileRef()
  {
    try { File.Delete (FilePath); }
    catch (Exception ex)
    {
      DeletionError = ex;
      FailedDeletions.Enqueue (this);   // Resurrection
    }
  }
}
```

Encolar el objeto en la colección estática `FailedDeletions` le otorga al objeto otro referente, lo que asegura que permanezca vivo hasta que finalmente sea desencolado.



ConcurrentQueue<T> es una versión segura para subprocesos de Queue<T> y se define en System.Collections.Concurrent (ver Capítulo 22). Hay un par de razones para usar una colección segura para subprocesos. En primer lugar, el CLR se reserva el derecho de ejecutar finalizadores en más de un subproceso de forma paralela. Esto significa que cuando se accede a un estado compartido, como una colección estática, debemos considerar la posibilidad de que dos objetos sean finalizados al mismo tiempo. En segundo lugar, en algún momento vamos a querer desencolar elementos de FailedDeletions para poder hacer algo con ellos. Esto también debe hacerse de manera segura para subprocesos, ya que podría ocurrir mientras el finalizador está encolando otro objeto en paralelo.

GC.ReRegisterForFinalize

El finalizador de un objeto resucitado no se ejecutará por segunda vez a menos que se llame a GC.ReRegisterForFinalize.

En el siguiente ejemplo, intentamos eliminar un archivo temporal en un finalizador (como en el ejemplo anterior). Pero si la eliminación falla, volvemos a registrar el objeto para intentarlo nuevamente en la siguiente recolección de basura:

```
public class TempFileRef
{
  public readonly string FilePath;
  int _deleteAttempt;

  public TempFileRef (string filePath) { FilePath = filePath; }

  ~TempFileRef()
  {
    try { File.Delete (FilePath); }
    catch
    {
      if (_deleteAttempt++ < 3) GC.ReRegisterForFinalize (this);
    }
  }
}
```

Después del tercer intento fallido, nuestro finalizador dejará de intentar eliminar el archivo sin mostrar ningún mensaje. Podríamos mejorar esto combinándolo con el ejemplo anterior, es decir, agregándolo a la cola FailedDeletions después del tercer fallo.

 Ten cuidado de llamar a ReRegisterForFinalize solo una vez en el método del finalizador. ¡Si lo llamas dos veces, el objeto se registrará dos veces y tendrá que pasar por dos finalizaciones más!

Cómo funciona el GC

El CLR estándar utiliza un GC generacional de marca y compactación que realiza la administración automática de memoria para objetos almacenados en el montón administrado. El GC se considera un GC de rastreo en el sentido de que no interfiere en cada acceso a un objeto, sino que se activa intermitentemente y rastrea el grafo de objetos almacenados en el montón administrado para determinar qué objetos se pueden considerar basura y, por lo tanto, recoger.

El GC inicia una recolección de basura al realizar una asignación de memoria (mediante la palabra clave new), ya sea después de haber asignado una cierta cantidad de memoria o en otros momentos para reducir la huella de memoria de la aplicación. Este proceso también se puede iniciar manualmente llamando a System.GC.Collect. Durante una recolección de basura, todos los hilos pueden quedar congelados (más sobre esto en la siguiente sección).

El GC comienza con sus referencias a objetos raíz y recorre el grafo de objetos, marcando todos los objetos que toca como alcanzables. Cuando este proceso se completa, todos los objetos que no se han marcado se consideran no utilizados y están sujetos a la recolección de basura. Los objetos no utilizados sin finalizadores se descartan inmediatamente; los objetos no utilizados con finalizadores se encolan para su procesamiento en el hilo del finalizador después de que se complete el GC. Estos objetos luego se vuelven elegibles para la recolección en el siguiente GC para la generación del objeto (a menos que se resuciten).

Los objetos "vivos" restantes se desplazan al comienzo del montón (compactación), liberando espacio para más objetos. Esta compactación sirve para dos propósitos: evita la fragmentación de memoria y permite que el GC emplee una estrategia muy simple al asignar nuevos objetos, que es siempre asignar memoria al final del montón. Esto evita la tarea potencialmente lenta de mantener una lista de segmentos de memoria libre.

Si no hay suficiente espacio para asignar memoria a un nuevo objeto después de la recolección de basura y el sistema operativo no puede conceder más memoria, se produce una excepción OutOfMemoryException.



Puede obtener información sobre el estado actual del montón administrado llamando a GC.GetGCMemoryInfo(). A partir de .NET 5, este método se ha mejorado para devolver datos relacionados con el rendimiento.

Técnicas de optimización

El GC incorpora diversas técnicas de optimización para reducir el tiempo de recolección de basura.

Colección generacional

La optimización más importante es que el GC es generacional. Esto aprovecha el hecho de que aunque muchos objetos se asignan y descartan rápidamente, ciertos objetos tienen una vida útil prolongada y, por lo tanto, no es necesario rastrearlos en cada recolección. Básicamente, el GC divide el montón administrado en tres generaciones. Los objetos que acaban de ser asignados están en Gen0, y los objetos que han sobrevivido a un ciclo de recolección están en Gen1; todos los demás objetos están en Gen2. Gen0 y Gen1 se conocen como generaciones efímeras (de corta duración).

El CLR mantiene la sección Gen0 relativamente pequeña (con un tamaño típico de unos pocos cientos de KB a unos pocos MB). Cuando la sección Gen0 se llena, el GC inicia una recolección de Gen0, lo cual ocurre con relativa frecuencia. El GC aplica un umbral de memoria similar a Gen1 (que actúa como un búfer para Gen2), por lo que las recolecciones de Gen1 también son relativamente rápidas y frecuentes. Sin embargo, las recolecciones completas que incluyen Gen2 son mucho más largas y, por lo tanto, ocurren con poca frecuencia. La Figura 12-2 muestra el efecto de una recolección completa.

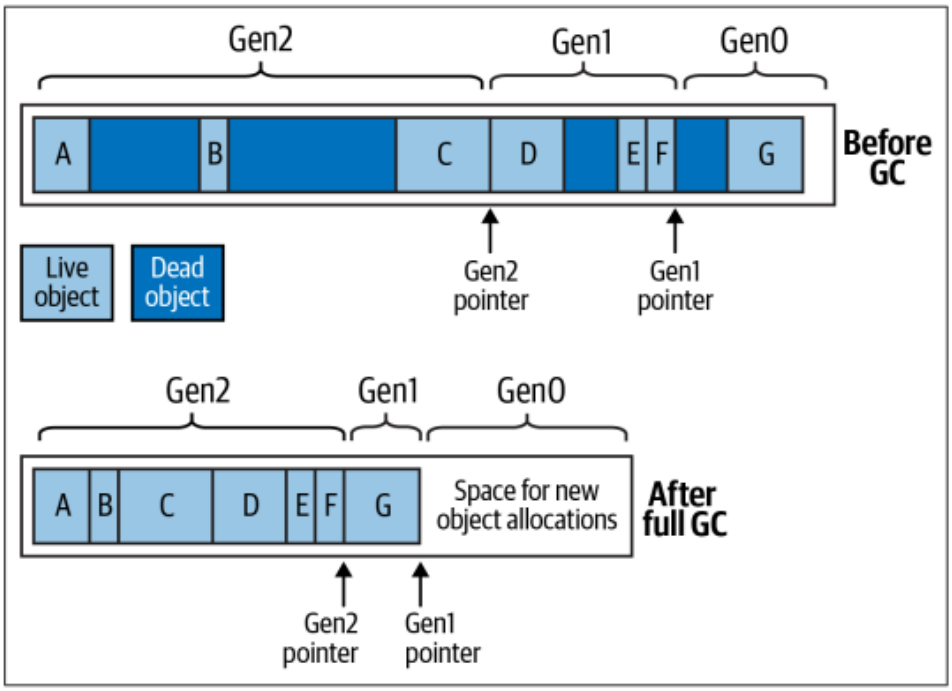


Figure 12-2. Heap generations

Para dar una idea aproximada, una recolección de Gen0 podría llevar menos de un milisegundo, lo cual no es suficiente para ser percibido en una aplicación típica. Sin embargo, una recolección completa podría llevar hasta 100 ms en un programa con grandes estructuras de objetos. Estas cifras dependen de numerosos factores y pueden variar considerablemente, especialmente en el caso de Gen2, cuyo tamaño no está limitado (a diferencia de Gen0 y Gen1). En resumen, los objetos de corta duración son muy eficientes en su uso del GC. Los StringBuilders creados en el siguiente método probablemente se recolectarían en un Gen0 rápido:

```
string Foo()
{
  var sb1 = new StringBuilder ("test");
  sb1.Append ("...");
  var sb2 = new StringBuilder ("test");
  sb2.Append (sb1.ToString());
  return sb2.ToString();
}
```

El Montón de Objetos Grandes (The Large Object Heap)

El GC utiliza un montón separado llamado Large Object Heap (LOH) para objetos más grandes que un umbral determinado (actualmente 85,000 bytes). Esto evita el costo de compactar objetos grandes y evita recolecciones excesivas de Gen0, ya que sin el LOH, asignar una serie de objetos de 16 MB podría desencadenar una recolección de Gen0 después de cada asignación.

Por defecto, el LOH no se somete a compactación, porque mover bloques grandes de memoria durante la recolección de basura sería demasiado costoso. Esto tiene dos consecuencias:

- Las asignaciones pueden ser más lentas, porque el GC no siempre puede simplemente asignar objetos al final del montón, también debe buscar espacios en el medio y esto requiere mantener una lista enlazada de bloques de memoria libre.

- El LOH está sujeto a fragmentación. Esto significa que la liberación de un objeto puede crear un hueco en el LOH que puede ser difícil de llenar más tarde. Por ejemplo, un hueco dejado por un objeto de 86,000 bytes solo puede ser llenado por un objeto de entre 85,000 y 86,000 bytes (a menos que esté adyacente a otro hueco).

Si anticipas un problema de fragmentación, puedes indicarle al GC que compacte el LOH en la próxima recolección de la siguiente manera:

```
GCSettings.LargeObjectHeapCompactionMode =
  GCLargeObjectHeapCompactionMode.CompactOnce;
```

Otra solución alternativa, si tu programa asigna frecuentemente matrices grandes, es utilizar el API de agrupación de matrices de .NET (consultar "Array Pooling" en la página 575).

El LOH también es no generacional: todos los objetos se tratan como Gen2.

Recopilación de estaciones de trabajo versus recopilación de servidores

.NET proporciona dos modos de recolección de basura: estaciones de trabajo y servidor. La estación de trabajo es la configuración predeterminada; puedes cambiar a servidor agregando lo siguiente al archivo .csproj de tu aplicación:

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

Al compilar el proyecto, esta configuración se escribe en el archivo .runtime config.json de la aplicación, donde es leído por el CLR:

```
"runtimeOptions": {
  "configProperties": {
    "System.GC.Server": true
  ...
```

Cuando se habilita la colección del servidor, el CLR asigna un montón de memoria y un GC separado para cada núcleo. Esto acelera la recolección pero consume memoria adicional y recursos de CPU (ya que cada núcleo requiere su propio hilo). Si la máquina está ejecutando muchos otros procesos con la colección del servidor habilitada, esto puede llevar a una sobreasignación de CPU, lo cual es particularmente perjudicial en estaciones de trabajo porque hace que todo el sistema operativo se sienta lento en general.

La colección del servidor solo está disponible en sistemas multinúcleo: en dispositivos de un solo núcleo (o máquinas virtuales de un solo núcleo), la configuración se ignora.

Recolección en segundo plano

Tanto en los modos de estación de trabajo como de servidor, el CLR habilita la recolección en segundo plano de forma predeterminada. Puedes deshabilitarla agregando lo siguiente al archivo .csproj de tu aplicación:

```
<PropertyGroup>
  <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
</PropertyGroup>
```

Al compilar, esta configuración se escribe en el archivo .runtimeconfig.json de la aplicación:

```
"runtimeOptions": {
  "configProperties": {
    "System.GC.Concurrent": false,
  ...
```

El GC debe bloquear (congelar) los hilos de ejecución durante ciertos períodos durante una recolección. La recolección en segundo plano minimiza estos períodos de latencia, lo que hace que tu aplicación sea más receptiva. Sin embargo, esto conlleva un ligero aumento en el consumo de CPU y memoria. Al deshabilitar la recolección en segundo plano, logras lo siguiente:

- Reducir ligeramente el uso de CPU y memoria.
- Aumentar las pausas (o latencia) cuando ocurre una recolección de basura.

La recolección en segundo plano funciona permitiendo que tu código de aplicación se ejecute en paralelo con una recolección de Gen2. (Las recolecciones de Gen0 y Gen1 se consideran suficientemente rápidas y no se benefician de este paralelismo).

La recolección en segundo plano es una versión mejorada de lo que antes se llamaba recolección concurrente: elimina una limitación que hacía que una recolección concurrente dejara de ser concurrente si la sección Gen0 se llenaba mientras se ejecutaba una recolección de Gen2. Esto permite que las aplicaciones que asignan memoria continuamente sean más receptivas.

Notificaciones de GC

Si deshabilitas la recolección en segundo plano, puedes solicitar que el GC te notifique justo antes de que ocurra una recolección completa (bloqueante). Esto está destinado a configuraciones de granja de servidores: la idea es que desvíes las solicitudes a otro servidor justo antes de una recolección. Luego, inicias la recolección de inmediato y esperas a que se complete antes de redirigir las solicitudes de nuevo a ese servidor.

Para iniciar las notificaciones, llama a GC.RegisterForFullGCNotification. Luego, inicia otro hilo (ver Capítulo 14) que primero llame a GC.WaitForFullGCApproach. Cuando este método devuelva un GCNotificationStatus que indique que se acerca una recolección, puedes redirigir las solicitudes a otros servidores y forzar una recolección manual (ver la siguiente sección). Luego llamas a GC.WaitForFullGCComplete: cuando este método devuelve un valor, la recolección está completa y puedes volver a aceptar solicitudes. Luego repites todo el ciclo.

Forzar la recolección de basura

Puedes forzar manualmente una recolección de basura en cualquier momento llamando a GC.Collect. Llamar a GC.Collect sin argumentos inicia una recolección completa. Si pasas un valor entero, solo se recolectarán las generaciones hasta ese valor, por lo que GC.Collect(0) realiza solo una recolección rápida de Gen0.

En general, obtienes el mejor rendimiento al permitir que el GC decida cuándo recolectar: forzar la recolección puede afectar el rendimiento al promover innecesariamente objetos de Gen0 a

Gen1 (y objetos de Gen1 a Gen2). También puede alterar la capacidad de autoajuste del GC, mediante la cual el GC ajusta dinámicamente los umbrales para cada generación para maximizar el rendimiento a medida que se ejecuta la aplicación.

Sin embargo, hay excepciones. El caso más común para intervenir es cuando una aplicación se queda inactiva durante un tiempo: un buen ejemplo es un servicio de Windows que realiza una actividad diaria (por ejemplo, verificar actualizaciones). Esta aplicación puede usar un System.Timers.Timer para iniciar la actividad cada 24 horas. Después de completar la actividad, no se ejecuta más código durante 24 horas, lo que significa que durante este período no se realizan asignaciones de memoria y, por lo tanto, el GC no tiene oportunidad de activarse. Cualquier memoria que el servicio consuma al realizar su actividad, la seguirá consumiendo durante las siguientes 24 horas, ¡incluso con un gráfico de objetos vacío! La solución es llamar a GC.Collect justo después de que se complete la actividad diaria.

Para asegurar la recolección de objetos cuya recolección se retrasa por finalizadores, toma el paso adicional de llamar a WaitForPendingFinalizers y volver a recolectar:

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

A menudo esto se hace en un bucle: la ejecución de los finalizadores puede liberar más objetos que ellos mismos tienen finalizadores.

Otro caso para llamar a GC.Collect es cuando estás probando una clase que tiene un finalizador.

Ajuste de la recolección de basura en tiempo de ejecución

La propiedad estática GCSettings.LatencyMode determina cómo el GC equilibra la latencia con la eficiencia general. Cambiar esto desde su valor predeterminado de Interactive a LowLatency o SustainedLowLatency instruye al CLR a favorecer recolecciones más rápidas (pero más frecuentes). Esto es útil si tu aplicación necesita responder muy rápidamente a eventos en tiempo real. Cambiar el modo a "Batch" maximiza el rendimiento a expensas de una potencial respuesta deficiente, lo cual es útil para el procesamiento por lotes.

"SustainedLowLatency" no es compatible si desactivas la recolección en segundo plano en el archivo ".runtimeconfig.json".

También puedes indicarle al CLR que suspenda temporalmente la recolección de basura llamando a GC.TryStartNoGCRegion y reanudarla con GC.EndNoGCRegion.

Presión de memoria

El tiempo de ejecución decide cuándo iniciar las recolecciones basándose en diversos factores, incluida la carga total de memoria en la máquina. Si tu programa asigna memoria no administrada (Capítulo 24), el tiempo de ejecución tendrá una percepción irrealmente optimista del uso de memoria porque el CLR solo conoce la memoria administrada. Puedes mitigar esto indicándole al CLR que asuma que se ha asignado una cantidad especificada de memoria no administrada; esto lo haces llamando a GC.AddMemoryPressure. Para deshacer esto (cuando se libera la memoria no administrada), llama a GC.RemoveMemoryPressure.

Agrupación de matrices

Si tu aplicación crea matrices con frecuencia, puedes evitar la mayor parte de la sobrecarga de la recolección de basura con la agrupación de matrices. La agrupación de matrices se introdujo en .NET Core 3 y funciona al "alquilar" una matriz, que luego devuelves a un grupo para su reutilización.

Para asignar una matriz, llama al método Rent en la clase ArrayPool del espacio de nombres System.Buffers, indicando el tamaño de la matriz que deseas:

```
int[] pooledArray = ArrayPool<int>.Shared.Rent (100);  // 100 bytes
```

Esto asigna una matriz de (al menos) 100 bytes desde el grupo global compartido de matrices. El administrador del grupo puede darte una matriz que sea más grande de lo que solicitaste (normalmente, asigna en potencias de 2).

Cuando hayas terminado con la matriz, llama a Return: esto libera la matriz al grupo, permitiendo que la misma matriz se alquile nuevamente:

```
ArrayPool<int>.Shared.Return (pooledArray);
```

Opcionalmente, puedes pasar un valor booleano indicando al administrador del grupo que borre la matriz antes de devolverla al grupo.

 Una limitación de la agrupación de matrices es que nada te impide seguir usando (ilegalmente) una matriz después de haberla devuelto, así que debes codificar con cuidado para evitar este escenario. Ten en cuenta que tienes el poder de romper no solo tu propio código, sino también otras APIs que utilizan la agrupación de matrices, como ASP.NET Core.

En lugar de usar el grupo de matrices compartido, puedes crear un grupo personalizado y alquilar desde él. Esto evita el riesgo de romper otras APIs, pero aumenta el uso general de memoria (ya que reduce las oportunidades de reutilización):

```
var myPool = ArrayPool<int>.Create();
int[] array = myPool.Rent (100);
...
```

Fugas de memoria administrada (Managed Memory Leaks)

En lenguajes no administrados como C++, debes recordar desasignar manualmente la memoria cuando un objeto ya no es necesario, de lo contrario, se producirá una fuga de memoria. En el mundo administrado, este tipo de error es imposible debido al sistema automático de recolección de basura del CLR.

No obstante, las aplicaciones .NET grandes y complejas pueden mostrar una forma más suave del mismo síndrome con el mismo resultado final: la aplicación consume cada vez más memoria a lo largo de su vida útil, hasta que eventualmente debe reiniciarse. La buena noticia es que las fugas de memoria administradas suelen ser más fáciles de diagnosticar y prevenir.

Las fugas de memoria administradas son causadas por objetos no utilizados que siguen vivos debido a referencias no utilizadas o olvidadas. Un candidato común son los controladores de eventos: estos mantienen una referencia al objeto objetivo (a menos que el objetivo sea un método estático). Por ejemplo, considera las siguientes clases:

```
class Host
{
  public event EventHandler Click;
}

class Client
{
  Host _host;
  public Client (Host host)
  {
    _host = host;
    _host.Click += HostClicked;
  }

  void HostClicked (object sender, EventArgs e) { ... }
}
```

La siguiente clase de prueba contiene un método que instancia 1,000 clientes:

```
class Test
{
  static Host _host = new Host();

  public static void CreateClients()
  {
    Client[] clients = Enumerable.Range (0, 1000)
     .Select (i => new Client (_host))
     .ToArray();

    // Do something with clients ...
  }
}
```

Es de esperar que después de que `CreateClients` termine de ejecutarse, los 1,000 objetos `Client` sean elegibles para la recolección. Desafortunadamente, cada cliente tiene otro referente: el objeto `_host`, cuyo evento `Click` ahora hace referencia a cada instancia de `Client`. Esto puede pasar desapercibido si el evento `Click` no se dispara, o si el método `HostClicked` no hace nada que atraiga la atención.

Una forma de solucionar esto es hacer que `Client` implemente `IDisposable` y, en el método `Dispose`, desvincular el controlador de eventos:

```
public void Dispose() { _host.Click -= HostClicked; }
```

Luego, los consumidores de `Client` deben llamar al método `Dispose` en las instancias cuando hayan terminado con ellas:

```
Array.ForEach (clients, c => c.Dispose());
```



En "Referencias débiles" en la página 579, describimos otra solución a este problema, que puede ser útil en entornos que tienden a no utilizar objetos desechables (un ejemplo es Windows Presentation Foundation [WPF]). De hecho, WPF ofrece una clase llamada `WeakEventManager` que utiliza un patrón que emplea referencias débiles.

Temporizadores (Timers)

Los temporizadores olvidados también pueden causar fugas de memoria (discutimos los temporizadores en el Capítulo 21). Hay dos escenarios distintos, dependiendo del tipo de temporizador. Veamos primero el temporizador en el espacio de nombres `System.Timers`. En el siguiente ejemplo, la clase `Foo` (cuando se instancia) llama al método `tmr_Elapsed` una vez por segundo:

```
using System.Timers;

class Foo
{
  Timer _timer;

  Foo()
  {
    _timer = new System.Timers.Timer { Interval = 1000 };
    _timer.Elapsed += tmr_Elapsed;
    _timer.Start();
  }

  void tmr_Elapsed (object sender, ElapsedEventArgs e) { ... }
}
```

¡Desafortunadamente, las instancias de `Foo` nunca pueden ser recolectadas por el recolector de basura! El problema es que el tiempo de ejecución mismo mantiene referencias a los temporizadores activos para que pueda disparar sus eventos `Elapsed`. Por lo tanto:

- El tiempo de ejecución mantendrá el objeto `_timer` vivo.
- El objeto `_timer` mantendrá vivo la instancia de `Foo` a través del controlador de eventos `tmr_Elapsed`.

La solución es obvia cuando te das cuenta de que Timer implementa IDisposable.

Al llamar a Dispose en el timer, se detiene y asegura que el tiempo de ejecución ya no haga referencia al objeto:

```
class Foo : IDisposable
{
  ...
  public void Dispose() { _timer.Dispose(); }
}
```



Una buena práctica es implementar IDisposable si cualquier campo de tu clase es asignado a un objeto que implementa IDisposable.

Los temporizadores de WPF y Windows Forms se comportan de la misma manera en relación a lo que se acaba de discutir.

Sin embargo, el temporizador en el espacio de nombres System.Threading es especial. .NET no mantiene referencias a temporizadores de subprocesos activos; en su lugar, hace referencia directamente a los delegados de devolución de llamada. Esto significa que si olvidas llamar a Dispose en un temporizador de subprocesos, se puede activar un finalizador que detendrá y desechará automáticamente el temporizador:

```
static void Main()
{
  var tmr = new System.Threading.Timer (TimerTick, null, 1000, 1000);
  GC.Collect();
  System.Threading.Thread.Sleep (10000);    // Wait 10 seconds
}

static void TimerTick (object notUsed) { Console.WriteLine ("tick"); }
```

Si este ejemplo se compila en modo "release" (depuración desactivada y optimizaciones habilitadas), el temporizador se recolectará y finalizará antes de tener la oportunidad de activarse ni una sola vez. Nuevamente, podemos solucionar esto llamando a Dispose en el temporizador cuando hayamos terminado de usarlo:

```
using (var tmr = new System.Threading.Timer (TimerTick, null, 1000, 1000))
{
  GC.Collect();
  System.Threading.Thread.Sleep (10000);    // Wait 10 seconds
}
```

La llamada implícita a tmr.Dispose al final del bloque using asegura que la variable `tmr` se "use" y, por lo tanto, no se considere muerta por el recolector de basura hasta el final del bloque. ¡Irónicamente, esta llamada a Dispose mantiene el objeto vivo durante más tiempo!

Diagnóstico de fugas de memoria (Diagnosing Memory Leaks)

La forma más fácil de evitar fugas de memoria administradas es monitorear proactivamente el consumo de memoria mientras se desarrolla una aplicación. Puedes obtener el consumo actual de memoria de un programa de la siguiente manera (el argumento true indica al GC que realice una recolección primero):

```
long memoryUsed = GC.GetTotalMemory (true);
```

Si estás practicando el desarrollo guiado por pruebas (TDD), una posibilidad es utilizar pruebas unitarias para verificar que la memoria se recupere según lo esperado. Si dicha verificación falla, solo necesitas examinar los cambios que has realizado recientemente.

Si ya tienes una aplicación grande con una fuga de memoria administrada, la herramienta windbg.exe puede ayudar a encontrarla. También hay herramientas gráficas más amigables como el CLR Profiler de Microsoft, el Memory Profiler de SciTech y el ANTS Memory Profiler de Red Gate.

El CLR también expone numerosos contadores de eventos para ayudar con la supervisión de recursos.

Referencias débiles (Weak References)

Ocasionalmente, es útil mantener una referencia a un objeto que es "invisible" para el recolector de basura en términos de mantener vivo el objeto. Esto se llama una referencia débil y se implementa mediante la clase System.WeakReference.

Para usar WeakReference, constrúyelo con un objeto objetivo:

```
var sb = new StringBuilder ("this is a test");
var weak = new WeakReference (sb);
Console.WriteLine (weak.Target);     // This is a test
```

Si un objetivo solo es referenciado por una o más referencias débiles, el recolector de basura considerará que el objetivo es elegible para la recolección. Cuando se recolecta el objetivo, la propiedad Target de WeakReference será nula:

```
var weak = GetWeakRef();
GC.Collect();
Console.WriteLine (weak.Target);   // (nothing)

WeakReference GetWeakRef () =>
  new WeakReference (new StringBuilder ("weak"));
```

Para evitar que el objetivo se recolecte entre comprobar si es nulo y consumirlo, asígnalo a una variable local:

```
var sb = (StringBuilder) weak.Target;
if (sb != null) { /* Do something with sb */ }
```

Después de que un objetivo se haya asignado a una variable local, tiene una raíz sólida y no puede ser recolectado mientras esa variable esté en uso.

La siguiente clase utiliza referencias débiles para realizar un seguimiento de todos los objetos Widget que se han instanciado, sin evitar que esos objetos sean recolectados:

```
class Widget
{
  static List<WeakReference> _allWidgets = new List<WeakReference>();
```

```
  public readonly string Name;

  public Widget (string name)
  {
    Name = name;
    _allWidgets.Add (new WeakReference (this));
  }

  public static void ListAllWidgets()
  {
    foreach (WeakReference weak in _allWidgets)
    {
      Widget w = (Widget)weak.Target;
      if (w != null) Console.WriteLine (w.Name);
    }
  }
}
```

La única advertencia con este sistema es que la lista estática crecerá con el tiempo, acumulando referencias débiles con objetivos nulos. Por lo tanto, es necesario implementar alguna estrategia de limpieza.

Referencias Débiles y Caché (Weak References and Caching)

Una forma de utilizar WeakReference es para almacenar en caché grandes gráficos de objetos. Esto permite almacenar datos intensivos en memoria de forma breve sin causar un consumo excesivo de memoria:

```
_weakCache = new WeakReference (...);   // _weakCache is a field
...
var cache = _weakCache.Target;
if (cache == null) { /* Re-create cache & assign it to _weakCache */ }
```

Esta estrategia puede ser solo moderadamente efectiva en la práctica, ya que tienes poco control sobre cuándo se ejecuta el recolector de basura (GC) y qué generación elige recolectar. En particular, si tu caché permanece en Gen0, puede ser recolectada en cuestión de microsegundos (y recuerda que el GC no solo recolecta cuando la memoria está baja, sino que recolecta regularmente bajo condiciones normales de memoria). Por lo tanto, como mínimo, deberías utilizar una caché de dos niveles, comenzando con referencias fuertes que conviertas en referencias débiles con el tiempo.

Referencias Débiles y Eventos (Weak References and Events)

Ya hemos visto cómo los eventos pueden causar pérdidas de memoria administradas. La solución más sencilla es evitar suscribirse en tales condiciones o implementar un método Dispose para cancelar la suscripción. Las referencias débiles ofrecen otra solución.

Imagina un delegado que solo mantiene referencias débiles a sus objetivos. Tal delegado no mantendría vivos a sus objetivos, a menos que esos objetivos tuvieran referencias independientes. Por supuesto, esto no evitaría que un delegado activado alcance un objetivo no referenciado en el lapso entre el momento en que el objetivo es elegible para la recolección y el momento en que el GC lo alcance. Para que dicha solución sea efectiva, tu código debe ser

robusto en ese escenario. Suponiendo que ese sea el caso, puedes implementar una clase de delegado débil de la siguiente manera:

```
public class WeakDelegate<TDelegate> where TDelegate : class
{
  class MethodTarget
  {
    public readonly WeakReference Reference;
    public readonly MethodInfo Method;

    public MethodTarget (Delegate d)
    {
      // d.Target will be null for static method targets:
      if (d.Target != null) Reference = new WeakReference (d.Target);
      Method = d.Method;
    }
  }

  List<MethodTarget> _targets = new List<MethodTarget>();

  public WeakDelegate()
  {
    if (!typeof (TDelegate).IsSubclassOf (typeof (Delegate)))
      throw new InvalidOperationException
        ("TDelegate must be a delegate type");
  }

  public void Combine (TDelegate target)
  {
    if (target == null) return;

    foreach (Delegate d in (target as Delegate).GetInvocationList())
      _targets.Add (new MethodTarget (d));
  }

  public void Remove (TDelegate target)
  {
    if (target == null) return;
    foreach (Delegate d in (target as Delegate).GetInvocationList())
    {
      MethodTarget mt = _targets.Find (w =>
        Equals (d.Target, w.Reference?.Target) &&
        Equals (d.Method.MethodHandle, w.Method.MethodHandle));

      if (mt != null) _targets.Remove (mt);
    }
  }

  public TDelegate Target
  {
    get
    {
      Delegate combinedTarget = null;

      foreach (MethodTarget mt in _targets.ToArray())
```

```
        {
          WeakReference wr = mt.Reference;

          // Static target || alive instance target
          if (wr == null || wr.Target != null)
          {
            var newDelegate = Delegate.CreateDelegate (
              typeof(TDelegate), wr?.Target, mt.Method);
              combinedTarget = Delegate.Combine (combinedTarget, newDelegate);
          }
          else
            _targets.Remove (mt);
        }

        return combinedTarget as TDelegate;
      }
      set
      {
        _targets.Clear();
        Combine (value);
      }
    }
  }
```

Este código ilustra varios puntos interesantes en C# y el CLR. En primer lugar, observa que verificamos que TDelegate sea un tipo de delegado en el constructor. Esto se debe a una limitación en C#: la siguiente restricción de tipo es ilegal porque C# considera a System.Delegate como un tipo especial para el cual no se admiten restricciones:

```
... where TDelegate : Delegate   // Compiler doesn't allow this
```

En su lugar, debemos elegir una restricción de clase y realizar una verificación en tiempo de ejecución en el constructor.

En los métodos Combine y Remove, realizamos la conversión de referencia de target a Delegate mediante el operador as en lugar del operador de conversión más común. Esto se debe a que C# no permite el operador de conversión con este tipo de parámetro, debido a una posible ambigüedad entre una conversión personalizada y una conversión de referencia. Luego llamamos a GetInvocationList porque estos métodos podrían ser llamados con delegados multicast, es decir, delegados con más de un destinatario de método.

En la propiedad Target, construimos un delegado multicast que combina todos los delegados referenciados por referencias débiles cuyos targets están vivos, eliminando las referencias restantes (muertas) de la lista para evitar que la lista _targets crezca indefinidamente. (Podríamos mejorar nuestra clase haciendo lo mismo en el método Combine; otra mejora sería agregar bloqueos para garantizar la seguridad en subprocesos [ver "Bloqueo y seguridad en subprocesos" en la página 616].) También permitimos delegados sin una referencia débil en absoluto; estos representan delegados cuyo objetivo es un método estático.

Lo siguiente ilustra cómo consumir este delegado en la implementación de un evento:

```
public class Foo
{
  WeakDelegate<EventHandler> _click = new WeakDelegate<EventHandler>();

  public event EventHandler Click
  {
    add { _click.Combine (value); } remove { _click.Remove (value); }
  }

  protected virtual void OnClick (EventArgs e)
    => _click.Target?.Invoke (this, e);
}
```

13 Diagnostics

Cuando las cosas van mal, es importante que haya información disponible para ayudar a diagnosticar el problema. Un entorno de desarrollo integrado (IDE) o un depurador pueden ser de gran ayuda en este sentido, pero generalmente solo están disponibles durante el desarrollo.

Después de que una aplicación se distribuye, la propia aplicación debe recopilar y registrar información de diagnóstico. Para cumplir con este requisito, .NET proporciona un conjunto de herramientas para registrar información de diagnóstico, monitorear el comportamiento de la aplicación, detectar errores en tiempo de ejecución e integrarse con herramientas de depuración si están disponibles.

Algunas herramientas y API de diagnóstico son específicas de Windows porque dependen de características del sistema operativo Windows. En un esfuerzo por evitar que las API específicas de la plataforma abarroten el BCL de .NET, Microsoft las ha incluido en paquetes NuGet separados que puedes referenciar de forma opcional. Hay más de una docena de paquetes específicos de Windows, que puedes referenciar todos a la vez con el paquete maestro "Microsoft.Windows.Compatibility".

Los tipos en este capítulo están definidos principalmente en el espacio de nombres System.Diagnostics.

Compilación condicional

Puedes compilar condicionalmente cualquier sección de código en C# con directivas de preprocesador. Las directivas de preprocesador son instrucciones especiales para el compilador que comienzan con el símbolo # (y, a diferencia de otras construcciones de C#, deben aparecer en una línea propia). Lógicamente, se ejecutan antes de que se realice la compilación principal (aunque en la práctica, el compilador las procesa durante la fase de análisis léxico). Las directivas de preprocesador para la compilación condicional son #if, #else, #endif y #elif.

La directiva #if instruye al compilador a ignorar una sección de código a menos que se haya definido un símbolo específico. Puedes definir un símbolo en el código fuente utilizando la directiva #define (en cuyo caso, el símbolo se aplica solo a ese archivo) o en el archivo .csproj utilizando un elemento <DefineConstants> (en cuyo caso, el símbolo se aplica a toda la compilación):

```
#define TESTMODE            // #define directives must be at top of file
                            // Symbol names are uppercase by convention.
using System;

class Program
{
  static void Main()
  {
#if TESTMODE
    Console.WriteLine ("in test mode!");     // OUTPUT: in test mode!
#endif
  }
}
```

Si eliminamos la primera línea, el programa se compilaría con la declaración Console.WriteLine completamente eliminada del ejecutable, como si estuviera comentada.

La declaración #else es análoga a la instrucción else de C#, y #elif es equivalente a #else seguido de #if. Los operadores ||, && y ! realizan operaciones de or, and y not:

```
#if TESTMODE && !PLAYMODE      // if TESTMODE and not PLAYMODE
  ...
```

Sin embargo, ten en cuenta que no estás construyendo una expresión C# ordinaria, y los símbolos en los que operas no tienen ninguna conexión con variables, ya sean estáticas u otras.

Puedes definir símbolos que se apliquen a todos los archivos de un ensamblado editando el archivo .csproj (o en Visual Studio, yendo a la pestaña Build en la ventana Propiedades del proyecto). El siguiente ejemplo define dos constantes, TESTMODE y PLAYMODE:

```
<PropertyGroup>
  <DefineConstants>TESTMODE;PLAYMODE</DefineConstants>
</PropertyGroup>
```

Si has definido un símbolo a nivel de ensamblado y luego quieres "undefinirlo" para un archivo en particular, puedes hacerlo utilizando la directiva #undef.

Compilación condicional versus flags de variables estáticas

En su lugar, podrías implementar el ejemplo anterior con un simple campo estático:

```
static internal bool TestMode = true;

static void Main()
{
  if (TestMode) Console.WriteLine ("in test mode!");
}
```

Esto tiene la ventaja de permitir la configuración en tiempo de ejecución. Entonces, ¿por qué elegir la compilación condicional? La razón es que la compilación condicional puede llevarte a lugares a los que los flags de variables no pueden, como los siguientes:

- Incluir condicionalmente un atributo.
- Cambiar el tipo declarado de una variable.
- Cambiar entre diferentes espacios de nombres o alias de tipos en una directiva using; por ejemplo:

```
using TestType =
  #if V2
     MyCompany.Widgets.GadgetV2;
  #else
     MyCompany.Widgets.Gadget;
  #endif
```

Incluso puedes realizar refactorizaciones importantes bajo una directiva de compilación condicional, lo que te permite cambiar instantáneamente entre versiones antiguas y nuevas, y escribir bibliotecas que se puedan compilar con múltiples versiones de tiempo de ejecución, aprovechando las últimas características cuando estén disponibles.

Otra ventaja de la compilación condicional es que el código de depuración puede hacer referencia a tipos en ensamblados que no están incluidos en la implementación.

El atributo Conditional

El atributo Conditional le indica al compilador que ignore las llamadas a una clase o método en particular si el símbolo especificado no está definido.

Para ver cómo esto es útil, supongamos que escribiste un método para registrar información de estado de la siguiente manera:

```
static void LogStatus (string msg)
{
  string logFilePath = ...
  System.IO.File.AppendAllText (logFilePath, msg + "\r\n");
}
```

Ahora imagina que quieres que esto se ejecute solo si el símbolo LOGGINGMODE está definido. La primera solución es envolver todas las llamadas a LogStatus con una directiva #if:

```
#if LOGGINGMODE
LogStatus ("Message Headers: " + GetMsgHeaders());
#endif
```

Esto da un resultado ideal, pero es tedioso. La segunda solución es colocar la directiva #if dentro del método LogStatus. Sin embargo, esto plantea problemas si LogStatus se llama de la siguiente manera:

```
LogStatus ("Message Headers: " + GetComplexMessageHeaders());
```

GetComplexMessageHeaders siempre se llamaría, lo que podría tener un impacto en el rendimiento.

Podemos combinar la funcionalidad de la primera solución con la conveniencia de la segunda mediante la adición del atributo Conditional (definido en System.Diagnostics) al método LogStatus:

```
[Conditional ("LOGGINGMODE")]
static void LogStatus (string msg)
{
  ...
}
```

Esto indica al compilador que trate las llamadas a LogStatus como si estuvieran envueltas en una directiva #if LOGGINGMODE. Si el símbolo no está definido, todas las llamadas a LogStatus se eliminan por completo durante la compilación, incluyendo las expresiones de evaluación de

argumentos. (Por lo tanto, se omitirán cualquier expresión con efectos secundarios). Esto funciona incluso si LogStatus y el llamador están en ensamblados diferentes.



Otro beneficio de [Conditional] es que la comprobación de condicionalidad se realiza cuando se compila el llamador, en lugar de cuando se compila el método llamado. Esto es beneficioso porque te permite escribir una biblioteca que contiene métodos como LogStatus y construir solo una versión de esa biblioteca.

El atributo Conditional se ignora en tiempo de ejecución, es simplemente una instrucción para el compilador.

Alternativas al atributo Conditional

El atributo Conditional es inútil si necesitas habilitar o deshabilitar dinámicamente la funcionalidad en tiempo de ejecución: en su lugar, debes utilizar un enfoque basado en variables. Esto plantea la pregunta de cómo evitar elegantemente la evaluación de argumentos al llamar a métodos de registro condicionales. Un enfoque funcional resuelve esto:

```
using System;
using System.Linq;

class Program
{
  public static bool EnableLogging;

  static void LogStatus (Func<string> message)
  {
    string logFilePath = ...
    if (EnableLogging)
      System.IO.File.AppendAllText (logFilePath, message() + "\r\n");
  }
}
```

Una expresión lambda te permite llamar a este método sin tener una sintaxis excesiva:

```
LogStatus ( () => "Message Headers: " + GetComplexMessageHeaders() );
```

Si EnableLogging es falso, GetComplexMessageHeaders no se evalúa.

Clases Debug y Trace

Las clases Debug y Trace son clases estáticas que proporcionan capacidades básicas de registro y aserciones. Ambas clases son muy similares; la principal diferencia radica en su uso previsto.

La clase Debug está destinada a compilaciones de depuración (debug builds), mientras que la clase Trace está destinada tanto a compilaciones de depuración como de lanzamiento (release builds). Para lograr esto:

```
All methods of the Debug class are defined with [Conditional("DEBUG")].
All methods of the Trace class are defined with [Conditional("TRACE")].
```

Esto significa que todas las llamadas que realices a Debug o Trace se eliminan por el compilador a menos que definas los símbolos DEBUG o TRACE. (Visual Studio proporciona casillas de verificación para definir estos símbolos en la pestaña de compilación de las Propiedades del proyecto y habilita el símbolo TRACE de forma predeterminada en nuevos proyectos).

Tanto la clase Debug como la clase Trace proporcionan los métodos Write, WriteLine y WriteIf. De forma predeterminada, estos envían mensajes a la ventana de salida del depurador:

```
Debug.Write     ("Data");
Debug.WriteLine (23 * 34);
int x = 5, y = 3;
Debug.WriteIf   (x > y, "x is greater than y");
```

La clase Trace también proporciona los métodos TraceInformation, TraceWarning y TraceError. La diferencia en el comportamiento entre estos métodos y los métodos Write depende de los TraceListeners activos (se cubre esto en "TraceListener" en la página 590).

Fail y Assert

Las clases Debug y Trace ambas proporcionan los métodos Fail y Assert. Fail envía el mensaje a cada TraceListener en la colección Listeners de la clase Debug o Trace (se cubre esto en la siguiente sección), que de forma predeterminada escribe el mensaje en la salida de depuración:

```
Debug.Fail ("File data.txt does not exist!");
```

Assert simplemente llama a Fail si el argumento bool es falso; esto se llama hacer una aserción y puede indicar un error en el código si se viola. Es opcional especificar un mensaje de fallo:

```
Debug.Assert (File.Exists ("data.txt"), "File data.txt does not exist!");
var result = ...
Debug.Assert (result != null);
```

Los métodos Write, Fail y Assert también tienen sobrecargas para aceptar una categoría de cadena además del mensaje, lo cual puede ser útil para procesar la salida.

Una alternativa a la aserción es lanzar una excepción si se cumple la condición opuesta. Esta es una práctica común al validar los argumentos de un método:

```
public void ShowMessage (string message)
{
  if (message == null) throw new ArgumentNullException ("message");
  ...
}
```

Tales "afirmaciones" se compilan incondicionalmente y son menos flexibles en el sentido de que no se puede controlar el resultado de una afirmación fallida a través de TraceListeners. Y técnicamente, no son afirmaciones. Una afirmación es algo que, si se viola, indica un error en el código del método actual. Lanzar una excepción basada en la validación de argumentos indica un error en el código del llamador.

TraceListener

La clase Trace tiene una propiedad estática llamada Listeners que devuelve una colección de instancias de TraceListener. Estos son responsables de procesar el contenido emitido por los métodos Write, Fail y Trace.

De manera predeterminada, la colección de Listeners incluye un único listener (Default TraceListener). El listener predeterminado tiene dos características principales:

- Cuando está conectado a un depurador como Visual Studio, los mensajes se escriben en la ventana de salida de depuración; de lo contrario, se ignora el contenido del mensaje.
- Cuando se llama al método Fail (o una afirmación falla), la aplicación se termina.

Puedes cambiar este comportamiento mediante la eliminación (opcional) del listener predeterminado y luego agregando uno o más propios. Puedes escribir trace listeners desde cero (mediante la creación de una subclase de TraceListener) o utilizar uno de los tipos predefinidos:

- TextWriterTraceListener escribe en un flujo (Stream) o en un TextWriter o agrega contenido a un archivo.
- EventLogTraceListener escribe en el registro de eventos de Windows (solo en Windows).
- EventProviderTraceListener escribe en el subsistema Event Tracing for Windows (ETW) (compatible con múltiples plataformas).

TextWriterTraceListener se subclasifica aún más en ConsoleTraceListener, DelimitedListTraceListener, XmlWriterTraceListener y EventSchemaTraceListener.

A continuación se muestra un ejemplo que borra el listener predeterminado de Trace y luego agrega tres listeners: uno que agrega contenido a un archivo, otro que escribe en la consola y otro que escribe en el registro de eventos de Windows:

```
// Clear the default listener:
Trace.Listeners.Clear();

// Add a writer that appends to the trace.txt file:
Trace.Listeners.Add (new TextWriterTraceListener ("trace.txt"));

 // Obtain the Console's output stream, then add that as a listener:
 System.IO.TextWriter tw = Console.Out;
 Trace.Listeners.Add (new TextWriterTraceListener (tw));

 // Set up a Windows Event log source and then create/add listener.
 // CreateEventSource requires administrative elevation, so this would
 // typically be done in application setup.
 if (!EventLog.SourceExists ("DemoApp"))
   EventLog.CreateEventSource ("DemoApp", "Application");

 Trace.Listeners.Add (new EventLogTraceListener ("DemoApp"));
```

En el caso del registro de eventos de Windows, los mensajes que se escriben con los métodos Write, Fail o Assert siempre se muestran como mensajes de "Información" en el visor de eventos de Windows. Sin embargo, los mensajes que se escriben a través de los métodos TraceWarning y TraceError se muestran como advertencias o errores.

TraceListener también tiene un filtro de tipo TraceFilter que puedes establecer para controlar si un mensaje se escribe en ese listener. Para hacer esto, puedes instanciar una de las subclases predefinidas (EventTypeFilter o SourceFilter) o crear una subclase de TraceFilter y anular el método ShouldTrace. Esto se puede usar para filtrar por categoría, por ejemplo.

TraceListener también define las propiedades IndentLevel e IndentSize para controlar la indentación, y la propiedad TraceOutputOptions para escribir datos adicionales:

```
TextWriterTraceListener tl = new TextWriterTraceListener (Console.Out);
tl.TraceOutputOptions = TraceOptions.DateTime | TraceOptions.Callstack;
```

Las TraceOutputOptions se aplican al utilizar los métodos Trace:

```
Trace.TraceWarning ("Orange alert");

DiagTest.vshost.exe Warning: 0 : Orange alert
     DateTime=2007-03-08T05:57:13.6250000Z
     Callstack=   at System.Environment.GetStackTrace(Exception e, Boolean
needFileInfo)
     at System.Environment.get_StackTrace()     at ...
```

Flushing and Closing Listeners

Algunos listeners, como TextWriterTraceListener, finalmente escriben en un flujo (stream) que está sujeto a almacenamiento en caché. Esto tiene dos implicaciones:

- Es posible que un mensaje no aparezca inmediatamente en el flujo de salida o en el archivo.
- Debes cerrar o al menos hacer un flush (vaciar) del listener antes de que tu aplicación finalice; de lo contrario, perderás lo que está en la caché (hasta 4 KB, de forma predeterminada, si estás escribiendo en un archivo).

Las clases Trace y Debug proporcionan los métodos estáticos Close y Flush, que llaman a Close o Flush en todos los listeners (lo que a su vez llama a Close o Flush en cualquier escritor o flujo subyacente de diagnóstico). Close llama implícitamente a Flush, cierra los manejadores de archivos y evita que se escriban más datos.

Como regla general, llama a Close antes de que finalice una aplicación y llama a Flush siempre que desees asegurarte de que los datos de mensaje actuales se escriban. Esto se aplica si estás utilizando listeners basados en flujo o archivo.

Trace y Debug también proporcionan una propiedad AutoFlush que, si es true, fuerza un Flush después de cada mensaje.



Es una buena práctica establecer AutoFlush en true en Debug y Trace si estás utilizando algún listener basado en flujo o archivo. De lo contrario, si ocurre una excepción no controlada o un error crítico, se pueden perder los últimos 4 KB de información de diagnóstico.

Integración con el depurador

A veces, es útil que una aplicación interactúe con un depurador si está disponible. Durante el desarrollo, el depurador suele ser tu entorno de desarrollo integrado (por ejemplo, Visual Studio); en la implementación, es más probable que el depurador sea una de las herramientas de depuración de nivel inferior, como WinDbg, Cordbg o MDbg.

Adjuntar y detener la ejecución

La clase estática Debugger en System.Diagnostics proporciona funciones básicas para interactuar con un depurador, como Break, Launch, Log e IsAttached.

Un depurador primero debe adjuntarse a una aplicación para depurarla. Si inicias una aplicación desde el IDE, esto ocurre automáticamente, a menos que lo solicites de otra manera (al elegir "Iniciar sin depuración"). Sin embargo, a veces es inconveniente o imposible iniciar una aplicación en modo de depuración dentro del IDE. Un ejemplo es una aplicación de servicio de Windows o (irónicamente) un diseñador de Visual Studio.

Una solución es iniciar la aplicación normalmente y luego, en tu IDE, elegir Depurar proceso. Sin embargo, esto no te permite establecer puntos de interrupción al comienzo de la ejecución del programa.

La solución alternativa es llamar a Debugger.Break desde tu aplicación. Este método inicia un depurador, se adjunta a él y suspende la ejecución en ese punto. (Launch hace lo mismo, pero sin suspender la ejecución). Después de que esté adjunto, puedes registrar mensajes directamente en la ventana de salida del depurador con el método Log.

Puedes verificar si estás adjunto a un depurador verificando la propiedad IsAttached.

Atributos del depurador

Los atributos DebuggerStepThrough y DebuggerHidden proporcionan sugerencias al depurador sobre cómo manejar el paso a través de un método, constructor o clase en particular.

DebuggerStepThrough solicita que el depurador ejecute una función sin ninguna interacción del usuario. Este atributo es útil en métodos generados automáticamente y en métodos de proxy que redirigen el trabajo real a un método en otro lugar. En este último caso, el depurador seguirá mostrando el método de proxy en la pila de llamadas si se establece un punto de interrupción dentro del método "real", a menos que también se agregue el atributo DebuggerHidden. Puedes combinar estos dos atributos en proxies para ayudar al usuario a centrarse en depurar la lógica de la aplicación en lugar de la infraestructura:

```
[DebuggerStepThrough, DebuggerHidden]
void DoWorkProxy()
{
  // setup...
  DoWork();
  // teardown...
}

void DoWork() {...}   // Real method...
```

Procesos y hilos de procesos

Describimos en la última sección del Capítulo 6 cómo usar Process.Start para iniciar un nuevo proceso. La clase Process también te permite consultar e interactuar con otros procesos que se ejecutan en el mismo o en otro equipo. La clase Process forma parte de .NET Standard 2.0, aunque sus características están restringidas para la plataforma UWP.

Examinar procesos en ejecución

Los métodos Process.GetProcessXXX recuperan un proceso específico por su nombre o identificador de proceso, o todos los procesos que se ejecutan en el equipo actual o en un equipo especificado. Esto incluye tanto procesos administrados como no administrados. Cada instancia de Process tiene una gran cantidad de propiedades que mapean estadísticas como el nombre, el ID, la prioridad, la utilización de memoria y procesador, los identificadores de ventana, etc. El siguiente ejemplo enumera todos los procesos en ejecución en el equipo actual:

```
foreach (Process p in Process.GetProcesses())
using (p)
{
  Console.WriteLine (p.ProcessName);
  Console.WriteLine ("   PID:      " + p.Id);
  Console.WriteLine ("   Memory:   " + p.WorkingSet64);
  Console.WriteLine ("   Threads:  " + p.Threads.Count);
}
```

Process.GetCurrentProcess devuelve el proceso actual.

Puedes terminar un proceso llamando a su método Kill.

Examinando hilos en un proceso

También puedes enumerar los hilos de otros procesos utilizando la propiedad Process.Threads. Sin embargo, los objetos que obtienes no son objetos System.Threading.Thread; son objetos ProcessThread y están destinados a tareas administrativas en lugar de tareas de sincronización. Un objeto ProcessThread proporciona información de diagnóstico sobre el hilo subyacente y te permite controlar algunos aspectos, como su prioridad y afinidad de procesador:

```
public void EnumerateThreads (Process p)
{
  foreach (ProcessThread pt in p.Threads)
  {
    Console.WriteLine (pt.Id);
    Console.WriteLine ("   State:    " + pt.ThreadState);
    Console.WriteLine ("   Priority: " + pt.PriorityLevel);
    Console.WriteLine ("   Started:  " + pt.StartTime);
    Console.WriteLine ("   CPU time: " + pt.TotalProcessorTime);
  }
}
```

StackTrace y StackFrame

Las clases StackTrace y StackFrame proporcionan una vista de solo lectura de una pila de llamadas de ejecución. Puedes obtener trazas de pila para el subproceso actual o un objeto Exception. Esta información es útil principalmente con fines de diagnóstico, aunque también se puede utilizar en programación (hackeos). StackTrace representa una pila de llamadas completa; StackFrame representa una sola llamada de método dentro de esa pila.



Si solo necesitas conocer el nombre y el número de línea del método que realiza la llamada, los atributos de información del llamante pueden proporcionar una alternativa más fácil y rápida. Cubrimos este tema en "Atributos de información del llamante" en la página 229.

Si instancias un objeto StackTrace sin argumentos o con un argumento bool, obtendrás una instantánea de la pila de llamadas del subproceso actual. El argumento bool, si es true, instruye a StackTrace que lea los archivos .pdb (proyecto de depuración) del ensamblado si están presentes, lo que te da acceso a los datos de nombre de archivo, número de línea y desplazamiento de columna. Los archivos de depuración del proyecto se generan cuando compilas con el interruptor /debug (depuración). (Visual Studio compila con este interruptor a menos que lo solicites de otra manera a través de la configuración avanzada de compilación).

Después de obtener un objeto StackTrace, puedes examinar un marco en particular llamando a GetFrame, o obtener todos los marcos utilizando GetFrames:

```
static void Main() { A (); }
static void A()    { B (); }
static void B()    { C (); }
static void C()
{
  StackTrace s = new StackTrace (true);

  Console.WriteLine ("Total frames:   " + s.FrameCount);
  Console.WriteLine ("Current method: " + s.GetFrame(0).GetMethod().Name);
  Console.WriteLine ("Calling method: " + s.GetFrame(1).GetMethod().Name);
  Console.WriteLine ("Entry method:   " + s.GetFrame
                                     (s.FrameCount-1).GetMethod().Name);
```

```
  Console.WriteLine ("Call Stack:");
  foreach (StackFrame f in s.GetFrames())
    Console.WriteLine (
      "  File: "   + f.GetFileName() +
      "  Line: "   + f.GetFileLineNumber() +
      "  Col: "    + f.GetFileColumnNumber() +
      "  Offset: " + f.GetILOffset() +
      "  Method: " + f.GetMethod().Name);
}
```

Aquí está la salida:

```
Total frames:   4
Current method: C
Calling method: B
Entry method: Main
Call stack:
  File: C:\Test\Program.cs  Line: 15  Col: 4  Offset: 7  Method: C
  File: C:\Test\Program.cs  Line: 12  Col: 22  Offset: 6  Method: B
  File: C:\Test\Program.cs  Line: 11  Col: 22  Offset: 6  Method: A
  File: C:\Test\Program.cs  Line: 10  Col: 25  Offset: 6  Method: Main
```



El desplazamiento del lenguaje intermedio (IL) indica el desplazamiento de la instrucción que se ejecutará a continuación, no la instrucción que se está ejecutando actualmente. Curiosamente, sin embargo, el número de línea y columna (si hay un archivo .pdb presente) suele indicar el punto de ejecución real.

Esto sucede porque el CLR hace todo lo posible por inferir el punto de ejecución real al calcular la línea y columna a partir del desplazamiento del IL. El compilador emite IL de tal manera que esto sea posible, incluyendo la inserción de instrucciones nop (sin operación) en el flujo de IL.

Sin embargo, al compilar con optimizaciones habilitadas, se desactiva la inserción de instrucciones nop, por lo que la pila de llamadas puede mostrar el número de línea y columna de la siguiente instrucción a ejecutar. Obtener una pila de llamadas útil se ve dificultado aún más por el hecho de que la optimización puede realizar otros trucos, como colapsar métodos completos.

Un atajo para obtener la información esencial de una StackTrace completa es llamar a ToString en ella. Aquí está el resultado:

```
at DebugTest.Program.C() in C:\Test\Program.cs:line 16
at DebugTest.Program.B() in C:\Test\Program.cs:line 12
at DebugTest.Program.A() in C:\Test\Program.cs:line 11
at DebugTest.Program.Main() in C:\Test\Program.cs:line 10
```

También puedes obtener la pila de llamadas para un objeto Exception (mostrando lo que llevó a la excepción) pasando la Exception al constructor de StackTrace.



La excepción ya tiene una propiedad StackTrace; sin embargo, esta propiedad devuelve una cadena simple, no un objeto StackTrace. Un objeto StackTrace es mucho más útil para registrar excepciones que ocurren después de la implementación, donde no hay archivos .pdb disponibles, porque puedes registrar el desplazamiento de IL en lugar de los números de línea y columna. Con un desplazamiento de IL y ildasm, puedes identificar con precisión dónde ocurrió un error dentro de un método.

Registros de eventos de Windows

La plataforma Win32 proporciona un mecanismo de registro centralizado en forma de registros de eventos de Windows.

Las clases Debug y Trace que usamos anteriormente escriben en un registro de eventos de Windows si registramos un EventLogTraceListener. Sin embargo, con la clase EventLog, puedes escribir directamente en un registro de eventos de Windows sin usar Trace o Debug. También puedes usar esta clase para leer y monitorear datos de eventos.



Escribir en el registro de eventos de Windows tiene sentido en una aplicación de servicio de Windows, porque si algo sale mal, no puedes mostrar una interfaz de usuario que dirija al usuario a algún archivo especial donde se haya escrito información de diagnóstico. Además, como es práctica común que los servicios escriban en el registro de eventos de Windows, este es el primer lugar donde un administrador probablemente buscará si tu servicio falla.

Existen tres registros de eventos de Windows estándar, identificados por estos nombres:

- Application
- System
- Security

El registro de Application es donde la mayoría de las aplicaciones suelen escribir.

Para escribir en un registro de eventos de Windows

1. Elige uno de los tres registros de eventos (generalmente Application).

2. Decide un nombre de origen y créalo si es necesario (la creación requiere permisos administrativos).

3. Llama a EventLog.WriteEntry con el nombre del registro, el nombre del origen y los datos del mensaje.

El nombre del origen es un nombre fácilmente identificable para tu aplicación. Debes registrar un nombre de origen antes de usarlo: el método CreateEventSource realiza esta función. Luego puedes llamar a WriteEntry:

```
const string SourceName = "MyCompany.WidgetServer";

// CreateEventSource requires administrative permissions, so this would
// typically be done in application setup.
if (!EventLog.SourceExists (SourceName))
  EventLog.CreateEventSource (SourceName, "Application");

EventLog.WriteEntry (SourceName,
  "Service started; using configuration file=...",
  EventLogEntryType.Information);
```

El tipo EventLogEntryType puede ser Information, Warning, Error, SuccessAudit o FailureAudit. Cada uno se muestra con un icono diferente en el visor de eventos de Windows. También puedes especificar opcionalmente una categoría e ID de evento (cada uno es un número de tu elección) y proporcionar datos binarios opcionales.

CreateEventSource también te permite especificar el nombre de una máquina: esto es para escribir en el registro de eventos de otra computadora, si tienes suficientes permisos.

Lectura del registro de eventos

Para leer un registro de eventos, instancia la clase EventLog con el nombre del registro al que deseas acceder y opcionalmente el nombre de otra computadora en la que se encuentre el registro. Cada entrada de registro se puede leer a través de la propiedad Entries de la colección:

```
EventLog log = new EventLog ("Application");

Console.WriteLine ("Total entries: " + log.Entries.Count);

EventLogEntry last = log.Entries [log.Entries.Count - 1];
Console.WriteLine ("Index:   " + last.Index);
Console.WriteLine ("Source:  " + last.Source);
Console.WriteLine ("Type:    " + last.EntryType);
Console.WriteLine ("Time:    " + last.TimeWritten);
Console.WriteLine ("Message: " + last.Message);
```

Puedes enumerar todos los registros para la computadora actual (u otra) a través del método estático EventLog.GetEventLogs() (esto requiere privilegios administrativos para acceso completo):

```
foreach (EventLog log in EventLog.GetEventLogs())
  Console.WriteLine (log.LogDisplayName);
```

Esto normalmente imprime, como mínimo, Application, Security y System.

Monitoreo del registro de eventos

Puedes recibir una alerta cada vez que se escribe una entrada en un registro de eventos de Windows a través del evento EntryWritten. Esto funciona para registros de eventos en la computadora local y se activa independientemente de la aplicación que registró el evento.

Para habilitar el monitoreo de registros, sigue estos pasos:

1. Crea una instancia de EventLog y establece su propiedad EnableRaisingEvents en true.

2. Maneja el evento EntryWritten.

Aquí tienes un ejemplo:

```
using (var log = new EventLog ("Application"))
{
  log.EnableRaisingEvents = true;
  log.EntryWritten += DisplayEntry;
  Console.ReadLine();
}

void DisplayEntry (object sender, EntryWrittenEventArgs e)
{
  EventLogEntry entry = e.Entry;
  Console.WriteLine (entry.Message);
}
```

Contadores de rendimiento



Los Contadores de rendimiento son una función exclusiva de Windows y requieren el paquete NuGet System.Diagnostics.PerformanceCounter. Si estás trabajando en Linux o macOS, consulta "Herramientas de diagnóstico multiplataforma" en la página 603 para conocer alternativas.

Los mecanismos de registro que hemos discutido hasta ahora son útiles para capturar información para análisis futuro. Sin embargo, para obtener información sobre el estado actual de una aplicación (o del sistema en general), se necesita un enfoque más en tiempo real. La solución de Win32 para esta necesidad es la infraestructura de monitoreo de rendimiento, que consta de un conjunto de contadores de rendimiento que el sistema y las aplicaciones exponen, y los complementos de la consola de administración de Microsoft (MMC) utilizados para monitorear estos contadores en tiempo real.

Los contadores de rendimiento se agrupan en categorías como "System", "Processor", ".NET CLR Memory", etc. A veces, estas categorías también se denominan "objetos de rendimiento" por las herramientas de interfaz gráfica de usuario (GUI). Cada categoría agrupa un conjunto relacionado de contadores de rendimiento que monitorean un aspecto del sistema o de la aplicación. Ejemplos de contadores de rendimiento en la categoría ".NET CLR Memory" incluyen "% Time in GC", "# Bytes in All Heaps" y "Allocated bytes/sec".

Cada categoría puede tener opcionalmente una o más instancias que se pueden monitorear de forma independiente. Por ejemplo, esto es útil en el contador de rendimiento "% Processor Time" en la categoría "Processor", que permite monitorear la utilización de la CPU. En una

máquina multiprocesador, este contador admite una instancia para cada CPU, lo que te permite monitorear la utilización de cada CPU de forma independiente.

Las siguientes secciones ilustran cómo realizar tareas comúnmente necesarias, como determinar qué contadores están expuestos, monitorear un contador y crear tus propios contadores para exponer información del estado de la aplicación.


La lectura de contadores de rendimiento o categorías puede requerir privilegios de administrador en la computadora local o de destino, dependiendo de lo que se acceda.

Enumeración de los contadores disponibles

El siguiente ejemplo enumera todos los contadores de rendimiento disponibles en la computadora. Para aquellos que tienen instancias, enumera los contadores para cada instancia:

```
PerformanceCounterCategory[] cats =
  PerformanceCounterCategory.GetCategories();

foreach (PerformanceCounterCategory cat in cats)
{
  Console.WriteLine ("Category: " + cat.CategoryName);

  string[] instances = cat.GetInstanceNames();
  if (instances.Length == 0)
  {
    foreach (PerformanceCounter ctr in cat.GetCounters())
      Console.WriteLine ("  Counter: " + ctr.CounterName);
  }
  else   // Dump counters with instances
  {
    foreach (string instance in instances)
    {
      Console.WriteLine ("  Instance: " + instance);
      if (cat.InstanceExists (instance))
        foreach (PerformanceCounter ctr in cat.GetCounters (instance))
          Console.WriteLine ("    Counter: " + ctr.CounterName);
    }
  }
}
```


¡El resultado tiene más de 10,000 líneas! También lleva tiempo ejecutarse debido a que PerformanceCounterCategory.InstanceExists tiene una implementación ineficiente. En un sistema real, querrías recuperar la información más detallada solo cuando sea necesario.

El siguiente ejemplo utiliza LINQ para recuperar solo los contadores de rendimiento de .NET y escribe el resultado en un archivo XML:

```
var x =
  new XElement ("counters",
    from PerformanceCounterCategory cat in
         PerformanceCounterCategory.GetCategories()
    where cat.CategoryName.StartsWith (".NET")
    let instances = cat.GetInstanceNames()

     select new XElement ("category",
       new XAttribute ("name", cat.CategoryName),
       instances.Length == 0
       ?
         from c in cat.GetCounters()
         select new XElement ("counter",
           new XAttribute ("name", c.CounterName))
       :
         from i in instances
         select new XElement ("instance", new XAttribute ("name", i),
           !cat.InstanceExists (i)
           ?
             null
           :
             from c in cat.GetCounters (i)
             select new XElement ("counter",
               new XAttribute ("name", c.CounterName))
         )
     )
   );
 x.Save ("counters.xml");
```

Lectura de datos del contador de rendimiento

Para recuperar el valor de un contador de rendimiento, instancia un objeto PerformanceCounter y luego llama al método NextValue o NextSample. NextValue devuelve un valor float simple; NextSample devuelve un objeto CounterSample que expone un conjunto más avanzado de propiedades, como CounterFrequency, TimeStamp, BaseValue y RawValue.

El constructor de PerformanceCounter toma un nombre de categoría, nombre del contador y una instancia opcional. Entonces, para mostrar la utilización actual del procesador para todas las CPU, harías lo siguiente:

```
using PerformanceCounter pc = new PerformanceCounter ("Processor",
                                                       "% Processor Time",
                                                       "_Total");
Console.WriteLine (pc.NextValue());
```

O para mostrar el consumo de memoria "real" (es decir, privada) del proceso actual:

```
string procName = Process.GetCurrentProcess().ProcessName;
using PerformanceCounter pc = new PerformanceCounter ("Process",
                                                      "Private Bytes",
                                                      procName);
Console.WriteLine (pc.NextValue());
```

PerformanceCounter no expone un evento ValueChanged, por lo que si deseas monitorear cambios, debes hacerlo mediante sondeo. En el siguiente ejemplo, sondeamos cada 200 ms, hasta que se indique que debemos salir mediante un EventWaitHandle:

```
// need to import System.Threading as well as System.Diagnostics

static void Monitor (string category, string counter, string instance,
                     EventWaitHandle stopper)


 {
   if (!PerformanceCounterCategory.Exists (category))
     throw new InvalidOperationException ("Category does not exist");

   if (!PerformanceCounterCategory.CounterExists (counter, category))
     throw new InvalidOperationException ("Counter does not exist");

   if (instance == null) instance = "";   // "" == no instance (not null!)
   if (instance != "" &&
       !PerformanceCounterCategory.InstanceExists (instance, category))
     throw new InvalidOperationException ("Instance does not exist");

   float lastValue = 0f;
   using (PerformanceCounter pc = new PerformanceCounter (category,
                                                          counter, instance))
     while (!stopper.WaitOne (200, false))
     {
       float value = pc.NextValue();
       if (value != lastValue)         // Only write out the value
       {                               // if it has changed.
         Console.WriteLine (value);
         lastValue = value;
       }
     }
 }
```

Aquí te muestro cómo podemos utilizar este método para monitorear simultáneamente la actividad del procesador y del disco duro:

```
EventWaitHandle stopper = new ManualResetEvent (false);

new Thread (() =>
  Monitor ("Processor", "% Processor Time", "_Total", stopper)
).Start();

new Thread (() =>
  Monitor ("LogicalDisk", "% Idle Time", "C:", stopper)
).Start();

Console.WriteLine ("Monitoring - press any key to quit");
Console.ReadKey();
stopper.Set();
```

Creando contadores y escribiendo datos de rendimiento

Antes de escribir datos en los contadores de rendimiento, necesitas crear una categoría de rendimiento y un contador. Debes crear la categoría de rendimiento junto con todos los contadores que le pertenecen en un solo paso, de la siguiente manera:

```
string category = "Nutshell Monitoring";

// We'll create two counters in this category:
string eatenPerMin = "Macadamias eaten so far";
string tooHard = "Macadamias deemed too hard";

if (!PerformanceCounterCategory.Exists (category))
{
  CounterCreationDataCollection cd = new CounterCreationDataCollection();

  cd.Add (new CounterCreationData (eatenPerMin,
          "Number of macadamias consumed, including shelling time",
          PerformanceCounterType.NumberOfItems32));

  cd.Add (new CounterCreationData (tooHard,
          "Number of macadamias that will not crack, despite much effort",
          PerformanceCounterType.NumberOfItems32));

  PerformanceCounterCategory.Create (category, "Test Category",
    PerformanceCounterCategoryType.SingleInstance, cd);
}
```

El nuevo contador aparecerá en la herramienta de monitorización de rendimiento de Windows cuando elijas "Agregar contadores". Si más adelante deseas definir más contadores en la misma categoría, primero debes eliminar la categoría anterior llamando a PerformanceCounterCategory.Delete.



La creación y eliminación de contadores de rendimiento requiere privilegios de administrador. Por esta razón, generalmente se realiza como parte de la configuración de la aplicación.

Después de crear un contador, puedes actualizar su valor mediante la creación de una instancia de PerformanceCounter, estableciendo ReadOnly en false y estableciendo RawValue. También puedes utilizar los métodos Increment e IncrementBy para actualizar el valor existente:

```
string category = "Nutshell Monitoring";
string eatenPerMin = "Macadamias eaten so far";

using (PerformanceCounter pc = new PerformanceCounter (category,
                                                       eatenPerMin, ""))
{
  pc.ReadOnly = false;
  pc.RawValue = 1000;
  pc.Increment();
  pc.IncrementBy (10);
  Console.WriteLine (pc.NextValue());    // 1011
}
```

La clase Stopwatch

La clase Stopwatch proporciona un mecanismo conveniente para medir tiempos de ejecución. Stopwatch utiliza el mecanismo de mayor resolución que el sistema operativo y el hardware proporcionan, que suele ser inferior a un microsegundo. (En contraste, DateTime.Now y Environment.TickCount tienen una resolución de aproximadamente 15 ms.)

Para usar Stopwatch, llama a StartNew: esto instancia un Stopwatch y lo pone en marcha. (Alternativamente, puedes instanciarlo manualmente y luego llamar a Start). La propiedad Elapsed devuelve el intervalo transcurrido como un TimeSpan:

```
Stopwatch s = Stopwatch.StartNew();
System.IO.File.WriteAllText ("test.txt", new string ('*', 30000000));
Console.WriteLine (s.Elapsed);       // 00:00:01.4322661
```

Stopwatch también expone una propiedad ElapsedTicks, que devuelve el número de "ticks" transcurridos como un long. Para convertir de ticks a segundos, divide por Stopwatch.Frequency. También existe una propiedad ElapsedMilliseconds, que suele ser la más conveniente.

Llamar a Stop congela Elapsed y ElapsedTicks. No hay actividad en segundo plano generada por un Stopwatch "en ejecución", por lo que llamar a Stop es opcional.

Herramientas de diagnóstico multiplataforma

En esta sección, describiremos brevemente las herramientas de diagnóstico multiplataforma disponibles para .NET:

dotnet-counters

> Proporciona una visión general del estado de una aplicación en ejecución, incluyendo el uso de memoria y CPU.

dotnet-trace

> Permite un seguimiento detallado del rendimiento y eventos.

dotnet-dump

> Permite obtener un volcado de memoria bajo demanda o después de un bloqueo.

Estas herramientas no requieren privilegios de administrador y son adecuadas tanto para entornos de desarrollo como de producción.

dotnet-counters

La herramienta dotnet-counters supervisa el uso de memoria y CPU de un proceso .NET y escribe los datos en la consola (o en un archivo).

Para instalar la herramienta, ejecuta lo siguiente desde una línea de comandos o terminal con dotnet en la ruta:

```
dotnet tool install --global dotnet-counters
```

Luego puedes iniciar la monitorización de un proceso de la siguiente manera:

```
dotnet-counters monitor System.Runtime --process-id <<ProcessID>>
```

System.Runtime indica que queremos monitorizar todos los contadores de la categoría System.Runtime. Puedes especificar tanto una categoría como un nombre de contador (el comando dotnet-counters list lista todas las categorías y contadores disponibles).

La salida se actualiza continuamente y se ve así:

```
Press p to pause, r to resume, q to quit.
    Status: Running

[System.Runtime]
    # of Assemblies Loaded                          63
    % Time in GC (since last GC)                     0
    Allocation Rate (Bytes / sec)              244,864
    CPU Usage (%)                                    6
    Exceptions / sec                                 0
    GC Heap Size (MB)                                8
    Gen 0 GC / sec                                   0
    Gen 0 Size (B)                             265,176
    Gen 1 GC / sec                                   0
    Gen 1 Size (B)                             451,552
    Gen 2 GC / sec                                   0
    Gen 2 Size (B)                                  24
    LOH Size (B)                             3,200,296
    Monitor Lock Contention Count / sec              0
    Number of Active Timers                          0
    ThreadPool Completed Work Items / sec           15
    ThreadPool Queue Length                          0
    ThreadPool Threads Count                         9
    Working Set (MB)                                52
```

Aquí están todos los comandos disponibles:

Commands	Purpose
`list`	Displays a list of counter names along with a description of each
`ps`	Displays a list of dotnet processes eligible for monitoring
`monitor`	Displays values of selected counters (periodically refreshed)
`collect`	Saves counter information to a file

Se admiten los siguientes parámetros:

Options/arguments	Purpose
`--version`	Displays the version of *dotnet-counters*.
`-h, --help`	Displays help about the program.
`-p, --process-id`	ID of dotnet process to monitor. Applies to the `monitor` and `collect` commands.
`--refresh-interval`	Sets the desired refresh interval in seconds. Applies to the `monitor` and `collect` commands.
`-o, --output`	Sets the output file name. Applies to the `collect` command.
`--format`	Sets the output format. Valid are *csv* or *json*. Applies to the `collect` command.

dotnet-trace

Las trazas son registros con marca de tiempo de eventos en tu programa, como la llamada a un método o una consulta a una base de datos. Las trazas también pueden incluir métricas de rendimiento y eventos personalizados, y pueden contener contexto local, como el valor de variables locales. Tradicionalmente, .NET Framework y frameworks como ASP.NET utilizaban ETW. En .NET 5, las trazas de la aplicación se escriben en ETW cuando se ejecutan en Windows y en LTTng en Linux.

Para instalar la herramienta, ejecuta el siguiente comando:

```
dotnet tool install --global dotnet-trace
```

Para comenzar a registrar los eventos de un programa, ejecuta el siguiente comando:

```
dotnet-trace collect --process-id <<ProcessId>>
```

Esto ejecuta dotnet-trace con el perfil predeterminado, que recopila eventos de CPU y tiempo de ejecución de .NET, y escribe en un archivo llamado trace.nettrace. Puedes especificar otros perfiles con el interruptor --profile: gc-verbose realiza un seguimiento de la recolección de basura y la asignación de objetos muestreada, y gc-collect realiza un seguimiento de la recolección de basura con una sobrecarga baja. El interruptor -o te permite especificar un nombre de archivo de salida diferente.

La salida predeterminada es un archivo .netperf, que se puede analizar directamente en una máquina Windows con la herramienta PerfView. Alternativamente, puedes indicar a dotnet-trace que cree un archivo compatible con Speedscope, que es un servicio de análisis en línea gratuito en https://speedscope.app. Para crear un archivo Speedscope (.speedscope.json), utiliza la opción --format speedscope.



Puedes descargar la última versión de PerfView desde https://github.com/microsoft/perfview. Es posible que la versión incluida en Windows 10 no admita archivos .netperf.

Los siguientes comandos son compatibles:

Commands	Purpose
`collect`	Starts recording counter information to a file.
`ps`	Displays a list of dotnet processes eligible for monitoring.
`list-profiles`	Lists prebuilt tracing profiles with a description of providers and filters in each.
`convert <file>`	Converts from the *nettrace* (*.netperf*) format to an alternative format. Currently, *speedscope* is the only target option.

Eventos personalizados de trazado (Custom trace events)

Tu aplicación puede emitir eventos personalizados al definir una EventSource personalizada:

```
[EventSource (Name = "MyTestSource")]
public sealed class MyEventSource : EventSource
{
  public static MyEventSource Instance = new MyEventSource ();


   MyEventSource() : base (EventSourceSettings.EtwSelfDescribingEventFormat)
   {
   }

   public void Log (string message, int someNumber)
   {
     WriteEvent (1, message, someNumber);
   }
 }
```

El método WriteEvent está sobrecargado para aceptar diversas combinaciones de tipos simples (principalmente cadenas y enteros). Puedes llamarlo de la siguiente manera:

```
MyEventSource.Instance.Log ("Something", 123);
```

Al llamar a dotnet-trace, debes especificar el nombre de cualquier origen de evento personalizado que desees registrar:

```
dotnet-trace collect --process-id <<ProcessId>> --providers MyTestSource
```

dotnet-dump

Un volcado de memoria, a veces llamado volcado principal, es una instantánea del estado de la memoria virtual de un proceso. Puedes generar un volcado de un proceso en ejecución bajo demanda o configurar el sistema operativo para generar un volcado cuando una aplicación se bloquea.

En Ubuntu Linux, el siguiente comando permite generar un volcado principal al producirse un bloqueo de la aplicación (los pasos necesarios pueden variar entre diferentes variantes de Linux):

```
ulimit -c unlimited
```

En Windows, utiliza regedit.exe para crear o editar la siguiente clave en la sección de la máquina local:

```
SOFTWARE\Microsoft\Windows\Windows Error Reporting\LocalDumps
```

Añade una clave con el mismo nombre que tu ejecutable (por ejemplo, foo.exe), y bajo esa clave, agrega las siguientes claves:

- DumpFolder (REG_EXPAND_SZ), con un valor que indique la ruta en la que deseas que se escriban los archivos de volcado.
- DumpType (REG_DWORD), con un valor de 2 para solicitar un volcado completo.
- (Opcionalmente) DumpCount (REG_DWORD), que indica el número máximo de archivos de volcado antes de que se elimine el más antiguo.

Para instalar la herramienta, ejecuta el siguiente comando:

```
dotnet tool install --global dotnet-dump
```

Después de haberla instalado, puedes iniciar un volcado bajo demanda (sin finalizar el proceso) de la siguiente manera:

```
dotnet-dump collect --process-id <<YourProcessId>>
```

El siguiente comando inicia una shell interactiva para analizar un archivo de volcado:

```
dotnet-dump analyze <<dumpfile>>
```

Si una excepción provocó el bloqueo de la aplicación, puedes utilizar el comando printexceptions (pe abreviado) para mostrar los detalles de esa excepción. La shell dotnet-dump admite numerosos comandos adicionales, que puedes listar con el comando help.

14 Concurrencia y Asincronía

La mayoría de las aplicaciones necesitan lidiar con más de una cosa sucediendo al mismo tiempo (concurrencia). En este capítulo, comenzamos con los requisitos esenciales, es decir, los conceptos básicos de los subprocesos (threads) y las tareas (tasks), y luego describimos en detalle los principios de la asincronía y las funciones asincrónicas de C#.

En el Capítulo 21, revisamos el multihilo con mayor detalle, y en el Capítulo 22, cubrimos el tema relacionado de la programación paralela.

Introducción

A continuación, se presentan los escenarios de concurrencia más comunes:

Escribir una interfaz de usuario receptiva

> En aplicaciones de Windows Presentation Foundation (WPF), móviles y Windows Forms, es necesario ejecutar tareas que consumen mucho tiempo en paralelo con el código que ejecuta la interfaz de usuario para mantener la capacidad de respuesta.

Permitir que las solicitudes se procesen simultáneamente

> En un servidor, las solicitudes de los clientes pueden llegar de manera concurrente y, por lo tanto, deben manejarse en paralelo para mantener la escalabilidad. Si utilizas ASP.NET Core o Web API, el tiempo de ejecución lo hace automáticamente por ti. Sin embargo, aún debes tener en cuenta el estado compartido (por ejemplo, el efecto de utilizar variables estáticas para la caché).

Programación paralela

> El código que realiza cálculos intensivos puede ejecutarse más rápido en computadoras multinúcleo/multiprocesador si la carga de trabajo se divide entre los núcleos (el Capítulo 22 se dedica a esto).

Ejecución especulativa

> En máquinas multinúcleo, a veces puedes mejorar el rendimiento al predecir algo que podría necesitarse y hacerlo de antemano. LINQPad utiliza esta técnica para acelerar la creación de nuevas consultas. Una variación es ejecutar varios algoritmos diferentes en paralelo que resuelven la misma tarea. El primero que termine "gana", esto es efectivo cuando no se puede saber de antemano qué algoritmo se ejecutará más rápido.

El mecanismo general mediante el cual un programa puede ejecutar código simultáneamente se llama multihilo (multithreading). El multihilo es compatible tanto con el CLR (Common Language Runtime) como con el sistema operativo, y es un concepto fundamental en la concurrencia. Por lo tanto, es esencial comprender los conceptos básicos de los subprocesos (threads) y, en particular, los efectos de los subprocesos en el estado compartido.

Threading

Un hilo (thread) es una ruta de ejecución que puede avanzar de forma independiente a otros hilos. Cada hilo se ejecuta dentro de un proceso del sistema operativo, el cual proporciona un entorno aislado en el que se ejecuta un programa. Con un programa de un solo hilo, solo se ejecuta un hilo en el entorno aislado del proceso, por lo que ese hilo tiene acceso exclusivo a él. Con un programa multihilo, varios hilos se ejecutan en un solo proceso, compartiendo el mismo entorno de ejecución (especialmente la memoria). Esto, en parte, es por qué el multihilo es útil: un hilo puede recuperar datos en segundo plano, mientras que otro hilo muestra los datos a medida que llegan. A estos datos se les conoce como estado compartido.

Creación de un hilo

Un programa cliente (Console, WPF, UWP o Windows Forms) comienza en un solo hilo que es creado automáticamente por el sistema operativo (el "hilo principal" o "main" thread). Aquí vive su vida como una aplicación de un solo hilo, a menos que hagas algo más, como crear más hilos directa o indirectamente.

Puedes crear y comenzar un nuevo hilo instanciando un objeto Thread y llamando a su método Start. El constructor más simple para Thread toma un delegado ThreadStart, que es un método sin parámetros que indica dónde debe comenzar la ejecución. Aquí tienes un ejemplo:

```
// NB: All samples in this chapter assume the following namespace imports:
using System;
using System.Threading;

Thread t = new Thread (WriteY);          // Kick off a new thread
t.Start();                               // running WriteY()

// Simultaneously, do something on the main thread.
for (int i = 0; i < 1000; i++) Console.Write ("x");

void WriteY()
{
  for (int i = 0; i < 1000; i++) Console.Write ("y");
}

 // Typical Output:
 xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyy
 yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx
 xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
 yyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 ...
```

El hilo principal crea un nuevo hilo "t" en el cual se ejecuta un método que imprime repetidamente el carácter "y". Simultáneamente, el hilo principal imprime repetidamente el carácter "x", como se muestra en la Figura 14-1. En una computadora de un solo núcleo, el sistema operativo debe asignar "segmentos" de tiempo a cada hilo (normalmente 20 ms en Windows) para simular la concurrencia, lo que resulta en bloques repetidos de "x" e "y". En una máquina multicore o multiprocesador, los dos hilos pueden ejecutarse genuinamente en paralelo (sujeto a la competencia de otros procesos activos en la computadora), aunque aún

obtienes bloques repetidos de "x" e "y" en este ejemplo debido a sutilezas en el mecanismo mediante el cual la consola maneja las solicitudes concurrentes.

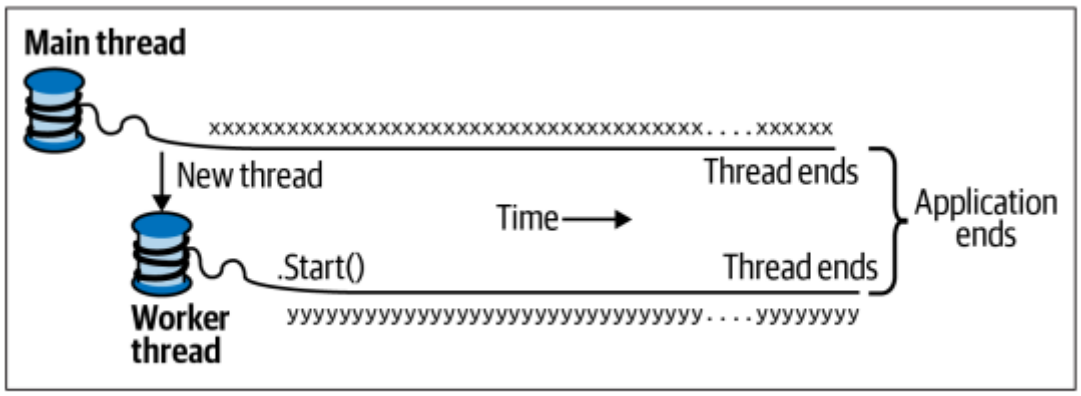


Figure 14-1. Starting a new thread



Un hilo se dice que es preempted en los puntos en los que su ejecución se intercala con la ejecución de código en otro hilo. El término a menudo surge al explicar por qué algo ha salido mal.

Después de que se inicia, la propiedad IsAlive de un hilo devuelve true hasta el momento en que el hilo finaliza. Un hilo finaliza cuando el delegado pasado al constructor del hilo termina de ejecutarse. Después de finalizar, un hilo no se puede reiniciar.

Cada hilo tiene una propiedad Name que puedes establecer para facilitar la depuración. Esto es especialmente útil en Visual Studio porque el nombre del hilo se muestra en la ventana Threads y en la barra de herramientas Debug Location. Puedes establecer el nombre de un hilo solo una vez; los intentos de cambiarlo más tarde generarán una excepción.

La propiedad estática `Thread.CurrentThread` te proporciona el hilo que se está ejecutando actualmente:

```
Console.WriteLine (Thread.CurrentThread.Name);
```

Join y Sleep

Puedes esperar a que otro hilo termine llamando a su método `Join`:

```
Thread t = new Thread (Go);
t.Start();
t.Join();
Console.WriteLine ("Thread t has ended!");

void Go() { for (int i = 0; i < 1000; i++) Console.Write ("y"); }
```

Esto imprime "y" 1,000 veces, seguido de "Thread t has ended!" inmediatamente después. Puedes incluir un tiempo de espera al llamar a Join, ya sea en milisegundos o como un objeto TimeSpan. Luego devuelve true si el hilo ha terminado o false si ha agotado el tiempo de espera.

`Thread.Sleep` pausa el hilo actual durante un período especificado:

```
Thread.Sleep (TimeSpan.FromHours (1));  // Sleep for 1 hour
Thread.Sleep (500);                     // Sleep for 500 milliseconds
```

`Thread.Sleep(0)` cede inmediatamente el tiempo de procesador actual del hilo, entregándolo voluntariamente a otros hilos. `Thread.Yield()` hace lo mismo, excepto que cede solo a hilos que se ejecutan en el mismo procesador.



`Sleep(0)` o `Yield` ocasionalmente son útiles en código de producción para ajustes avanzados de rendimiento. También son excelentes herramientas de diagnóstico para ayudar a descubrir problemas de seguridad de hilos: si insertar `Thread.Yield()` en cualquier parte de tu código rompe el programa, casi con seguridad tienes un error.

Mientras se espera en un Sleep o Join, el hilo está bloqueado.

Bloqueo

Un hilo se considera bloqueado cuando su ejecución se pausa por algún motivo, como cuando se está durmiendo (Sleeping) o esperando a que otro hilo termine mediante Join. Un hilo bloqueado cede inmediatamente su tiempo de procesador y, a partir de entonces, no consume tiempo del procesador hasta que se satisface su condición de bloqueo. Puedes verificar si un hilo está bloqueado a través de su propiedad `ThreadState`:

```
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;
```



El estado de un hilo (ThreadState) es una enumeración de banderas que combina tres "capas" de datos de manera bitwise. Sin embargo, la mayoría de los valores son redundantes, no utilizados o están obsoletos. El siguiente método de extensión simplifica un ThreadState a uno de los cuatro valores útiles: Unstarted, Running, WaitSleepJoin y Stopped:

```
public static ThreadState Simplify (this ThreadState ts)
{
  return ts & (ThreadState.Unstarted |
               ThreadState.WaitSleepJoin |
               ThreadState.Stopped);
}
```

La propiedad ThreadState es útil para fines de diagnóstico, pero no es adecuada para la sincronización, ya que el estado de un hilo puede cambiar entre la comprobación del ThreadState y la acción basada en esa información.

Cuando un hilo se bloquea o se desbloquea, el sistema operativo realiza un cambio de contexto. Esto conlleva una pequeña sobrecarga, generalmente de uno o dos microsegundos.

I/O-bound frente a compute-bound

Una operación que pasa la mayor parte de su tiempo esperando a que algo suceda se denomina I/O-bound. Un ejemplo es descargar una página web o llamar a Console.ReadLine. (Las operaciones I/O-bound típicamente involucran entrada o salida, pero esto no es un requisito estricto: Thread.Sleep también se considera I/O-bound). En contraste, una operación que pasa la mayor parte de su tiempo realizando un trabajo intensivo en CPU se denomina compute-bound.

Bloqueo frente a giro (spinning)

Una operación I/O-bound funciona de dos maneras: o bien espera de forma sincrónica en el hilo actual hasta que la operación se complete (como Console.ReadLine, Thread.Sleep o Thread.Join), o bien opera de forma asíncrona, disparando un callback cuando la operación se complete en el futuro (más sobre esto más adelante).

Las operaciones I/O-bound que esperan de forma sincrónica pasan la mayor parte de su tiempo bloqueando un hilo. También pueden "girar" en un bucle periódicamente:

```
while (DateTime.Now < nextStartTime)
  Thread.Sleep (100);
```

Dejando de lado que hay formas mejores de hacer esto (como temporizadores o constructos de señalización), otra opción es que un hilo gire continuamente:

```
while (DateTime.Now < nextStartTime);
```

En general, esto es muy ineficiente en términos de uso del procesador: desde el punto de vista del CLR (Common Language Runtime) y el sistema operativo, el hilo está realizando un cálculo importante y, por lo tanto, se asignan recursos en consecuencia. En efecto, hemos convertido lo que debería ser una operación I/O-bound en una operación compute-bound.



Existen un par de matices con respecto al giro (spinning) versus bloqueo (blocking). En primer lugar, girar brevemente puede ser efectivo cuando se espera que una condición se cumpla pronto (quizás dentro de unos pocos microsegundos) porque evita la sobrecarga y latencia de un cambio de contexto. .NET proporciona métodos y clases especiales para ayudar en esto; consulta el suplemento en línea "SpinLock y SpinWait" para más información.

En segundo lugar, el bloqueo no tiene un costo cero. Esto se debe a que cada hilo ocupa alrededor de 1 MB de memoria mientras está activo y causa una sobrecarga administrativa continua para el CLR y el sistema operativo. Por esta razón, el bloqueo puede ser problemático en el contexto de programas intensivos en operaciones de E/S (entrada/salida) que necesitan manejar cientos o miles de operaciones concurrentes. En su lugar, dichos programas deben utilizar un enfoque basado en callbacks, cediendo completamente el hilo mientras esperan. Este es (en parte) el propósito de los patrones asíncronos que se discuten más adelante.

Estado local versus estado compartido

El CLR asigna a cada hilo su propia pila de memoria para que las variables locales se mantengan separadas. En el siguiente ejemplo, definimos un método con una variable local y luego llamamos al método simultáneamente en el hilo principal y en un hilo recién creado:

```
new Thread (Go).Start();      // Call Go() on a new thread
Go();                         // Call Go() on the main thread

void Go()
{
  // Declare and use a local variable - 'cycles'
  for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

Se crea una copia separada de la variable `cycles` en la pila de memoria de cada hilo, por lo que la salida es predeciblemente 10 signos de interrogación.

Los hilos comparten datos si tienen una referencia común al mismo objeto o variable:

```
bool _done = false;

new Thread (Go).Start();
Go();

void Go()
{
   if (!_done) { _done = true; Console.WriteLine ("Done"); }
}
```

Ambos hilos comparten la variable `_done`, por lo que se imprime "Done" una vez en lugar de dos veces.

Las variables locales capturadas por una expresión lambda también pueden ser compartidas:

```
bool done = false;
ThreadStart action = () =>
{

  if (!done) { done = true; Console.WriteLine ("Done"); }
};
new Thread (action).Start();
action();
```

Más comúnmente, los campos (fields) se utilizan para compartir datos entre hilos. En el siguiente ejemplo, ambos hilos llaman a Go() en la misma instancia de ThreadTest, por lo que comparten el mismo campo _done:

```
var tt = new ThreadTest();
new Thread (tt.Go).Start();
tt.Go();

class ThreadTest
{
  bool _done;

  public void Go()
  {
    if (!_done) { _done = true; Console.WriteLine ("Done"); }
  }
}
```

Los campos estáticos ofrecen otra forma de compartir datos entre hilos:

```
  class ThreadTest
  {
    static bool _done;    // Static fields are shared between all threads
                          // in the same process.
    static void Main()
    {
      new Thread (Go).Start();
      Go();
    }

    static void Go()
    {
      if (!_done) { _done = true; Console.WriteLine ("Done"); }
    }
  }
```

Los cuatro ejemplos ilustran otro concepto clave: el de la seguridad en los hilos (o más bien, la falta de ella). El resultado es en realidad indeterminado: es posible (aunque poco probable) que se imprima "Done" dos veces. Sin embargo, si intercambiamos el orden de las instrucciones en el método Go, las probabilidades de que se imprima "Done" dos veces aumentan drásticamente:

```
static void Go()
{
  if (!_done) { Console.WriteLine ("Done"); _done = true; }
}
```

El problema radica en que un hilo puede estar evaluando la instrucción if exactamente al mismo tiempo que el otro hilo está ejecutando la instrucción WriteLine, antes de que haya tenido la oportunidad de establecer _done en true.



Nuestro ejemplo ilustra una de las muchas formas en que el estado compartido escribible puede introducir el tipo de errores intermitentes por los cuales la programación multihilo es conocida. A continuación, veremos cómo solucionar nuestro programa mediante el uso de bloqueos; sin embargo, es mejor evitar el estado compartido en la medida de lo posible. Más adelante veremos cómo los patrones de programación asíncrona ayudan en este aspecto.

Bloqueos y seguridad en hilos



Los bloqueos y la seguridad en hilos son temas amplios. Para obtener una discusión completa, consulta "Bloqueo exclusivo" en la página 866 y "Bloqueos y seguridad en hilos" en la página 874.

Podemos solucionar el ejemplo anterior obteniendo un bloqueo exclusivo al leer y escribir en el campo compartido. C# proporciona la declaración lock para este propósito:

```
class ThreadSafe
{
  static bool _done;
  static readonly object _locker = new object();

  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }

  static void Go()
  {
    lock (_locker)
    {
      if (!_done) { Console.WriteLine ("Done"); _done = true; }
    }
  }
}
```

Cuando dos hilos compiten simultáneamente por un bloqueo (que puede ser sobre cualquier objeto de tipo referencia; en este caso, _locker), un hilo espera o se bloquea hasta que el bloqueo esté disponible. En este caso, se asegura de que solo un hilo pueda ingresar a su bloque de código a la vez y "Done" se imprimirá solo una vez. El código que está protegido de esta manera, evitando la indeterminación en un contexto multihilo, se denomina "thread-safe".



Incluso el acto de incrementar automáticamente una variable no es seguro en hilos: la expresión x++ se ejecuta en el procesador subyacente como operaciones distintas de lectura-incremento-escritura. Entonces, si dos hilos ejecutan x++ al mismo tiempo fuera de un bloqueo, la variable puede terminar incrementándose solo una vez en lugar de dos veces (o peor aún, x podría estar fragmentada, terminando con una mezcla bit a bit del contenido antiguo y nuevo, bajo ciertas condiciones).

El uso de bloqueos no es una solución infalible para la seguridad en hilos; es fácil olvidar poner un bloqueo alrededor del acceso a un campo y el uso de bloqueos puede crear problemas propios, como el bloqueo mutuo. Un buen ejemplo de cuándo se puede utilizar un bloqueo es alrededor del acceso a una caché en memoria compartida para objetos de base de datos de acceso frecuente en una aplicación ASP.NET. Este tipo de aplicación es simple de implementar

correctamente y no existe la posibilidad de bloqueo mutuo. Se proporciona un ejemplo en "Seguridad en hilos en servidores de aplicaciones" en la página 877 del Capítulo 21.

Pasar datos a un hilo

A veces, es posible que desees pasar argumentos al método de inicio del hilo. La forma más sencilla de hacerlo es con una expresión lambda que llame al método con los argumentos deseados:

```
Thread t = new Thread ( () => Print ("Hello from t!") );
t.Start();

void Print (string message) => Console.WriteLine (message);
```

Con este enfoque, puedes pasar cualquier cantidad de argumentos al método. Incluso puedes envolver toda la implementación en una expresión lambda con múltiples declaraciones:

```
new Thread (() =>
{
  Console.WriteLine ("I'm running on another thread!");
  Console.WriteLine ("This is so easy!");
}).Start();
```

Una técnica alternativa (y menos flexible) es pasar un argumento al método Start del hilo:

```
Thread t = new Thread (Print);
t.Start ("Hello from t!");

void Print (object messageObj)
{
  string message = (string) messageObj;   // We need to cast here
  Console.WriteLine (message);
}
```

Esto funciona porque el constructor de Thread está sobrecargado para aceptar dos delegados:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart (object obj);
```

Expresiones lambda y variables capturadas

Como vimos, una expresión lambda es la forma más conveniente y potente de pasar datos a un hilo. Sin embargo, debes tener cuidado de no modificar accidentalmente variables capturadas después de iniciar el hilo. Por ejemplo, considera el siguiente código:

```
for (int i = 0; i < 10; i++)
  new Thread (() => Console.Write (i)).Start();
```

¡La salida es no determinista! Aquí tienes un resultado típico:

```
0223557799
```

El problema es que la variable `i` se refiere a la misma ubicación de memoria durante toda la duración del bucle. Por lo tanto, cada hilo llama a `Console.Write` en una variable cuyo valor puede cambiar mientras se está ejecutando. La solución es usar una variable temporal de la siguiente manera:

```
for (int i = 0; i < 10; i++)
{
  int temp = i;
  new Thread (() => Console.Write (temp)).Start();
}
```

Cada uno de los dígitos del 0 al 9 se escribe exactamente una vez. (El orden aún es indefinido porque los hilos pueden comenzar en tiempos indeterminados).



Esto es análogo al problema que describimos en "Variables capturadas" en la página 410. El problema se trata tanto de las reglas de C# para capturar variables en bucles `for` como de la concurrencia en hilos.

La variable `temp` ahora es local en cada iteración del bucle. Por lo tanto, cada hilo captura una ubicación de memoria diferente y no hay problema. Podemos ilustrar el problema en el código anterior de manera más sencilla con el siguiente ejemplo:

```
string text = "t1";
Thread t1 = new Thread ( () => Console.WriteLine (text) );

text = "t2";
Thread t2 = new Thread ( () => Console.WriteLine (text) );

t1.Start(); t2.Start();
```

Debido a que ambas expresiones lambda capturan la misma variable `text`, se imprime "t2" dos veces.

Manejo de excepciones

Cualquier bloque try/catch/finally en vigor cuando se crea un hilo no tiene relevancia para el hilo cuando comienza a ejecutarse. Considera el siguiente programa:

```
try
{
  new Thread (Go).Start();
}
catch (Exception ex)
{
  // We'll never get here!
  Console.WriteLine ("Exception!");
}

void Go() { throw null; }   // Throws a NullReferenceException
```

La declaración try/catch en este ejemplo es ineficaz y el hilo recién creado se verá afectado por una NullReferenceException no controlada. Este comportamiento tiene sentido cuando se considera que cada hilo tiene una ruta de ejecución independiente.

La solución es mover el control de excepciones al método Go:

```
new Thread (Go).Start();

void Go()
{
  try
  {
    ...
    throw null;    // The NullReferenceException will get caught below
    ...
  }
  catch (Exception ex)
  {
    // Typically log the exception and/or signal another thread
    // that we've come unstuck
    ...
  }
}
```

Es necesario tener un control de excepciones en todos los métodos de entrada de hilos en aplicaciones de producción, al igual que en el hilo principal (generalmente a un nivel superior en la pila de ejecución). Una excepción no controlada hace que toda la aplicación se cierre, mostrando un cuadro de diálogo no deseado.



Al escribir bloques de control de excepciones como estos, rara vez se ignora el error. Normalmente, se registran los detalles de la excepción. En una aplicación de cliente, se podría mostrar un cuadro de diálogo que permita al usuario enviar automáticamente esos detalles a un servidor web. Luego, se podría optar por reiniciar la aplicación, ya que es posible que una excepción inesperada deje el programa en un estado inválido.

Manejo centralizado de excepciones

En las aplicaciones de WPF, UWP y Windows Forms, se puede suscribir a eventos de manejo de excepciones globales, como Application.DispatcherUnhandledException y Application.ThreadException, respectivamente. Estos eventos se activan después de una excepción no controlada en cualquier parte del programa que se llame a través del bucle de mensajes (lo cual abarca todo el código que se ejecuta en el hilo principal mientras la aplicación está activa). Esto es útil para registrar y reportar errores (aunque no se activará para excepciones no controladas en hilos de trabajo que se creen). El manejo de estos eventos evita que el programa se cierre, aunque se puede optar por reiniciar la aplicación para evitar la posible corrupción del estado que puede surgir de (o que llevó a) la excepción no controlada.

Hilos de Primer Plano versus Hilos de Fondo (Foreground Versus Background Threads)

Por defecto, los hilos que creas explícitamente son hilos de primer plano (foreground threads). Los hilos de primer plano mantienen viva la aplicación mientras cualquiera de ellos esté en ejecución, mientras que los hilos de fondo (background threads) no lo hacen. Después de que todos los hilos de primer plano finalicen, la aplicación termina y cualquier hilo de fondo que siga en ejecución se termina abruptamente.



El estado de primer plano o de fondo de un hilo no tiene relación con su prioridad (asignación de tiempo de ejecución).

Puedes consultar o cambiar el estado de fondo de un hilo utilizando su propiedad IsBackground:

```
static void Main (string[] args)
{
  Thread worker = new Thread ( () => Console.ReadLine() );
  if (args.Length > 0) worker.IsBackground = true;
  worker.Start();
}
```

Si este programa se ejecuta sin argumentos, el hilo de trabajo asume el estado de primer plano y esperará en la instrucción ReadLine a que el usuario presione Enter. Mientras tanto, el hilo principal sale, pero la aplicación sigue ejecutándose porque hay un hilo de primer plano activo. Por otro lado, si se pasa un argumento a Main(), el hilo de trabajo se asigna al estado de fondo y el programa sale casi de inmediato cuando el hilo principal termina (finalizando la instrucción ReadLine).

Cuando un proceso termina de esta manera, se evitan los bloques finally en la pila de ejecución de los hilos de fondo. Si tu programa utiliza bloques finally (o using) para realizar tareas de limpieza, como eliminar archivos temporales, puedes evitar esto esperando explícitamente a los hilos de fondo al salir de la aplicación, ya sea uniéndote al hilo o mediante un mecanismo de señalización (consultar "Señalización" en la página 621). En ambos casos, debes especificar un tiempo de espera para poder abandonar un hilo renegado en caso de que se niegue a finalizar; de lo contrario, tu aplicación no se cerrará sin que el usuario tenga que pedir ayuda al Administrador de tareas (o en Unix, al comando kill).

Los hilos de primer plano no requieren este tratamiento, pero debes tener cuidado de evitar errores que podrían hacer que el hilo no finalice. Una causa común de fallos en la salida adecuada de las aplicaciones es la presencia de hilos de primer plano activos.

Prioridad del hilo

La propiedad Priority de un hilo determina cuánto tiempo de ejecución se le asigna en relación con otros hilos activos en el sistema operativo, en la siguiente escala:

```
enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }
```

Esto se vuelve relevante cuando varios hilos están activos al mismo tiempo. Debes tener cuidado al elevar la prioridad de un hilo, ya que puede acaparar los recursos de otros hilos. Si deseas que

un hilo tenga una prioridad más alta que los hilos de otros procesos, también debes elevar la prioridad del proceso utilizando la clase Process en System.Diagnostics:

```
using Process p = Process.GetCurrentProcess();
p.PriorityClass = ProcessPriorityClass.High;
```

Esto puede funcionar bien para procesos no relacionados con la interfaz de usuario que realizan un trabajo mínimo y requieren baja latencia (capacidad de responder rápidamente) en su trabajo. Sin embargo, en aplicaciones intensivas en cómputo (especialmente aquellas con una interfaz de usuario), elevar la prioridad del proceso puede afectar negativamente a otros procesos, ralentizando toda la computadora.

Señalización

A veces, necesitas que un hilo espere hasta recibir notificaciones de otros hilos. Esto se conoce como señalización. La forma más simple de señalización es el ManualResetEvent. Llamar a WaitOne en un ManualResetEvent bloquea el hilo actual hasta que otro hilo "abra" la señal llamando a Set. En el siguiente ejemplo, iniciamos un hilo que espera en un ManualResetEvent. Permanece bloqueado durante dos segundos hasta que el hilo principal lo señaliza:

```
var signal = new ManualResetEvent (false);

new Thread (() =>
{
  Console.WriteLine ("Waiting for signal...");
  signal.WaitOne();
  signal.Dispose();
  Console.WriteLine ("Got signal!");
}).Start();

Thread.Sleep(2000);
signal.Set();        // “Open” the signal
```

Después de llamar a Set, la señal permanece abierta; puedes cerrarla nuevamente llamando a Reset. ManualResetEvent es una de varias construcciones de señalización proporcionadas por CLR; las cubrimos en detalle en el Capítulo 21.

Threading en aplicaciones de cliente enriquecidas

En aplicaciones de WPF, UWP y Windows Forms, ejecutar operaciones de larga duración en el hilo principal hace que la aplicación no responda porque el hilo principal también procesa el bucle de mensajes que realiza el renderizado y maneja los eventos del teclado y el mouse.

Un enfoque popular es iniciar hilos "trabajadores" para operaciones que consumen mucho tiempo. El código en un hilo trabajador ejecuta una operación que requiere mucho tiempo y luego actualiza la interfaz de usuario cuando se completa. Sin embargo, todas las aplicaciones de cliente enriquecidas tienen un modelo de threading en el cual los elementos y controles de la interfaz de usuario solo pueden ser accedidos desde el hilo que los creó (normalmente el hilo principal de la interfaz de usuario). Violar esto puede causar un comportamiento impredecible o lanzar una excepción.

Por lo tanto, cuando deseas actualizar la interfaz de usuario desde un hilo trabajador, debes enviar la solicitud al hilo de la interfaz de usuario (término técnico: marshal). La forma de hacer esto a nivel bajo es la siguiente (más adelante, discutiremos otras soluciones que se basan en esto):

- En WPF, llama a BeginInvoke o Invoke en el objeto Dispatcher del elemento.
- En aplicaciones UWP, llama a RunAsync o Invoke en el objeto Dispatcher.
- En Windows Forms, llama a BeginInvoke o Invoke en el control.

Todos estos métodos aceptan un delegado que hace referencia al método que deseas ejecutar. BeginInvoke/RunAsync funcionan encolando el delegado en la cola de mensajes del hilo de la interfaz de usuario (la misma cola que maneja eventos de teclado, mouse y temporizadores). Invoke hace lo mismo, pero luego se bloquea hasta que el mensaje haya sido leído y procesado por el hilo de la interfaz de usuario. Debido a esto, Invoke te permite obtener un valor de retorno del método. Si no necesitas un valor de retorno, BeginInvoke/RunAsync son preferibles ya que no bloquean al llamador y no introducen la posibilidad de un bloqueo (ver "Bloqueos" en la página 872).



Puedes imaginar que cuando llamas a Application.Run, se ejecuta el siguiente seudocódigo:

```
while (!thisApplication.Ended)
{
  wait for something to appear in message queue
  Got something: what kind of message is it?
    Keyboard/mouse message -> fire an event handler
    User BeginInvoke message -> execute delegate
    User Invoke message -> execute delegate & post result
}
```

Es este tipo de bucle el que permite que un hilo trabajador envíe un delegado para su ejecución en el hilo de la interfaz de usuario.

Para demostrarlo, supongamos que tenemos una ventana de WPF que contiene un cuadro de texto llamado "txtMessage", cuyo contenido queremos que un hilo trabajador actualice después de realizar una tarea que consume mucho tiempo (que simularemos llamando a Thread.Sleep). Así es cómo lo haríamos:

```
partial class MyWindow : Window
{
  public MyWindow()
  {
    InitializeComponent();
    new Thread (Work).Start();
  }

  void Work()
  {
    Thread.Sleep (5000);           // Simulate time-consuming task
    UpdateMessage ("The answer");
  }

  void UpdateMessage (string message)
  {

    Action action = () => txtMessage.Text = message;
    Dispatcher.BeginInvoke (action);
  }
}
```

Múltiples hilos de interfaz de usuario

Es posible tener múltiples hilos de interfaz de usuario si cada uno de ellos posee ventanas diferentes. El escenario principal es cuando tienes una aplicación con múltiples ventanas de nivel superior, a menudo llamada aplicación de interfaz de documento único (SDI, por sus siglas en inglés), como Microsoft Word. Cada ventana SDI típicamente se muestra como una "aplicación" separada en la barra de tareas y está en su mayoría aislada, funcionalmente, de otras ventanas SDI. Al asignar a cada ventana su propio hilo de interfaz de usuario, se puede lograr que cada ventana sea más receptiva en relación con las demás.

Ejecutar esto resulta en la aparición inmediata de una ventana receptiva. Cinco segundos después, actualiza el cuadro de texto. El código es similar para Windows Forms, excepto que llamamos al método BeginInvoke (del formulario):

```
void UpdateMessage (string message)
{
  Action action = () => txtMessage.Text = message;
  this.BeginInvoke (action);
}
```

Contextos de sincronización

En el espacio de nombres System.ComponentModel, hay una clase llamada SynchronizationContext que permite la generalización del enlace de hilos.

Las API de cliente rico para aplicaciones móviles y de escritorio (UWP, WPF y Windows Forms) definen e instancian subclases de SynchronizationContext, que se pueden obtener a través de la propiedad estática SynchronizationContext.Current (mientras se ejecuta en un hilo de interfaz de usuario). Capturar esta propiedad te permite luego "publicar" en controles de interfaz de usuario desde un hilo de trabajo:

```
partial class MyWindow : Window
{
  SynchronizationContext _uiSyncContext;

  public MyWindow()
  {
    InitializeComponent();
    // Capture the synchronization context for the current UI thread:
    _uiSyncContext = SynchronizationContext.Current;
    new Thread (Work).Start();
  }

  void Work()
  {
    Thread.Sleep (5000);           // Simulate time-consuming task

      UpdateMessage ("The answer");
    }

    void UpdateMessage (string message)
    {
      // Marshal the delegate to the UI thread:
      _uiSyncContext.Post (_ => txtMessage.Text = message, null);
    }
  }
```

Esto es útil porque la misma técnica funciona con todas las API de interfaz de usuario de cliente rico.

Llamar a `Post` es equivalente a llamar a `BeginInvoke` en un `Dispatcher` o `Control`; también existe un método `Send` que es equivalente a `Invoke`.

El grupo de subprocesos (Thread Pool)

Cuando se inicia un hilo, se gastan unos pocos cientos de microsegundos organizando cosas como una nueva pila de variables locales. El grupo de subprocesos reduce esta sobrecarga al tener un grupo de hilos pre-creados y reciclables. El grupo de subprocesos es esencial para la programación paralela eficiente y la concurrencia de granularidad fina; permite que las operaciones cortas se ejecuten sin verse abrumadas por los costos de inicio del hilo.

Hay algunas cosas de las que debes tener cuidado al usar hilos del grupo de subprocesos:

- No puedes establecer el nombre de un hilo del grupo de subprocesos, lo que dificulta la depuración (aunque puedes adjuntar una descripción al depurar en la ventana de hilos de Visual Studio).
- Los hilos del grupo de subprocesos siempre son hilos en segundo plano.

- Bloquear los hilos del grupo de subprocesos puede degradar el rendimiento (ver "Higiene en el grupo de subprocesos" en la página 625).

Puedes cambiar la prioridad de un hilo del grupo de subprocesos; se restaurará a la normalidad cuando se libere de nuevo al grupo.

Puedes determinar si estás ejecutando actualmente en un hilo del grupo de subprocesos a través de la propiedad `Thread.CurrentThread.IsThreadPoolThread`.

Ingresar al grupo de subprocesos

La forma más sencilla de ejecutar explícitamente algo en un hilo del grupo de subprocesos es utilizar `Task.Run` (lo cubrimos en más detalle en la sección siguiente):

```
// Task is in System.Threading.Tasks
Task.Run (() => Console.WriteLine ("Hello from the thread pool"));
```

Dado que las tareas no existían antes de .NET Framework 4.0, una alternativa común es llamar a `ThreadPool.QueueUserWorkItem`:

```
ThreadPool.QueueUserWorkItem (notUsed => Console.WriteLine ("Hello"));
```



Esto se utiliza implícitamente en el grupo de subprocesos:

- Servidores de aplicaciones ASP.NET Core y Web API.
- Temporizadores System.Timers.Timer y System.Threading.Timer.
- Los constructos de programación paralela que describimos en el Capítulo 22.
- La clase BackgroundWorker (heredada).

Higiene en el grupo de subprocesos

El grupo de subprocesos tiene otra función, que es asegurar que un exceso temporal de trabajo con uso intensivo de CPU no cause una sobresuscripción de la CPU. La sobresuscripción es la condición en la que hay más hilos activos que núcleos de CPU, y el sistema operativo tiene que realizar divisiones de tiempo entre los hilos. La sobresuscripción afecta el rendimiento porque las divisiones de tiempo requieren cambios de contexto costosos y pueden invalidar las cachés de CPU que se han vuelto esenciales para ofrecer un rendimiento a los procesadores modernos.

El CLR evita la sobresuscripción en el grupo de subprocesos mediante la encolación de tareas y la limitación de su inicio. Comienza ejecutando tantas tareas concurrentes como núcleos de hardware haya y luego ajusta el nivel de concurrencia mediante un algoritmo de escalada, ajustando continuamente la carga de trabajo en una dirección particular. Si mejora el rendimiento, continúa en la misma dirección (de lo contrario, revierte). Esto asegura que siempre siga la curva de rendimiento óptima, incluso ante la actividad de otros procesos en el equipo.

La estrategia del CLR funciona mejor si se cumplen dos condiciones:

- Los elementos de trabajo son principalmente de corta duración (< 250 ms, o idealmente < 100 ms) para que el CLR tenga muchas oportunidades de medir y ajustar.
- Las tareas que pasan la mayor parte de su tiempo bloqueadas no dominan el grupo de subprocesos.

El bloqueo es problemático porque da al CLR la falsa idea de que está cargando la CPU. El CLR es lo suficientemente inteligente como para detectarlo y compensarlo (inyectando más hilos en el grupo), aunque esto puede hacer que el grupo sea vulnerable a una sobresuscripción posterior. También puede introducir latencia porque el CLR limita la velocidad a la que inyecta nuevos hilos, especialmente al principio de la vida de una aplicación (más en sistemas operativos cliente donde favorece un menor consumo de recursos).

Mantener una buena higiene en el grupo de subprocesos es especialmente relevante cuando se desea utilizar completamente la CPU (por ejemplo, a través de las API de programación paralela en el Capítulo 22).

Las tareas (Tasks)

Un hilo es una herramienta de bajo nivel para crear concurrencia y, como tal, tiene limitaciones, especialmente las siguientes:

- Aunque es fácil pasar datos a un hilo que se inicia, no hay una forma sencilla de obtener un "valor de retorno" de un hilo al que te unes (Join). Necesitas configurar algún tipo de campo compartido. Y si la operación lanza una excepción, atrapar y propagar esa excepción también es complicado.
- No puedes indicarle a un hilo que inicie algo más cuando haya terminado; en su lugar, debes unirte a él (bloqueando tu propio hilo en el proceso).

Estas limitaciones desalientan la concurrencia de grano fino; es decir, dificultan la composición de operaciones concurrentes más grandes mediante la combinación de operaciones más pequeñas (algo esencial para la programación asíncrona que se analiza en las secciones siguientes). Esto a su vez lleva a una mayor dependencia de la sincronización manual (bloqueo, señalización, etc.) y los problemas asociados con ella.

El uso directo de hilos también tiene implicaciones en el rendimiento que se discutieron en "El grupo de subprocesos" en la página 624. Y si necesitas ejecutar cientos o miles de operaciones de E/S concurrentes, un enfoque basado en hilos consume cientos o miles de megabytes de memoria puramente en la sobrecarga de hilos.

La clase Task ayuda con todos estos problemas. En comparación con un hilo, una tarea es una abstracción de nivel superior: representa una operación concurrente que puede o no respaldarse mediante un hilo. Las tareas son componibles (puedes encadenarlas mediante el uso de continuaciones). Pueden utilizar el grupo de subprocesos para reducir la latencia de inicio y, con un TaskCompletionSource, pueden emplear un enfoque de devolución de llamada que evita por completo los hilos mientras espera operaciones de E/S.

Los tipos de tareas se introdujeron en el Framework 4.0 como parte de la biblioteca de programación paralela. Sin embargo, desde entonces se han mejorado (mediante el uso de

awaiters) para funcionar igual de bien en escenarios de concurrencia más generales y son los tipos subyacentes de las funciones asíncronas de C#.

 En esta sección, ignoramos las características de las tareas que están dirigidas específicamente a la programación paralela; las cubrimos en su lugar en el Capítulo 22.

Iniciar una tarea

La forma más sencilla de iniciar una tarea respaldada por un hilo es mediante el método estático Task.Run (la clase Task se encuentra en el espacio de nombres System.Threading.Tasks). Simplemente pasa un delegado Action:

```
Task.Run (() => Console.WriteLine ("Foo"));
```

 Las tareas utilizan hilos del grupo de subprocesos (thread pool) de forma predeterminada, que son hilos en segundo plano (background threads). Esto significa que cuando el hilo principal termina, también lo hacen las tareas que creaste. Por lo tanto, para ejecutar estos ejemplos desde una aplicación de consola, debes bloquear el hilo principal después de iniciar la tarea (por ejemplo, esperando a que la tarea termine o llamando a Console.ReadLine):

```
Task.Run (() => Console.WriteLine ("Foo"));
Console.ReadLine();
```

En los ejemplos de acompañamiento del libro para LINQPad, se omite Console.ReadLine porque el proceso de LINQPad mantiene los hilos en segundo plano vivos.

Llamar a Task.Run de esta manera es similar a iniciar un hilo de la siguiente manera (excepto por las implicaciones del grupo de subprocesos que discutiremos en breve):

```
new Thread (() => Console.WriteLine ("Foo")).Start();
```

Task.Run devuelve un objeto Task que podemos utilizar para monitorear su progreso, al igual que un objeto Thread. (Sin embargo, ten en cuenta que no llamamos a Start después de llamar a Task.Run porque este método crea tareas "activas" (hot tasks); en su lugar, puedes utilizar el constructor de Task para crear tareas "inactivas" (cold tasks), aunque rara vez se hace en la práctica).

Puedes rastrear el estado de ejecución de una tarea a través de su propiedad Status.

Wait

Llamar a Wait en una tarea bloquea hasta que se complete y es equivalente a llamar a Join en un hilo:

```
Task task = Task.Run (() =>
{
  Thread.Sleep (2000);
  Console.WriteLine ("Foo");
});
Console.WriteLine (task.IsCompleted);  // False
task.Wait();  // Blocks until task is complete
```

Wait te permite especificar opcionalmente un tiempo de espera y un token de cancelación para finalizar la espera de forma anticipada (consulta "Cancelación" en la página 659).

Tareas de larga duración

De forma predeterminada, el CLR ejecuta tareas en hilos del grupo de subprocesos, lo cual es ideal para trabajos de cómputo de corta duración. Para operaciones más largas y bloqueantes (como nuestro ejemplo anterior), puedes evitar el uso de un hilo del grupo de subprocesos de la siguiente manera:

```
Task task = Task.Factory.StartNew (() => ...,
                                   TaskCreationOptions.LongRunning);
```



Ejecutar una tarea de larga duración en un hilo del grupo de subprocesos no causará problemas; es cuando ejecutas múltiples tareas de larga duración en paralelo (especialmente aquellas que se bloquean) que el rendimiento puede verse afectado. En ese caso, generalmente existen mejores soluciones que utilizar TaskCreationOptions.LongRunning:

- Si las tareas están limitadas por E/S, TaskCompletionSource y las funciones asíncronas te permiten implementar concurrencia con devoluciones de llamada (continuaciones) en lugar de hilos.
- Si las tareas están limitadas por cómputo, una cola productor/consumidor te permite controlar la concurrencia para esas tareas, evitando que otros hilos y procesos se queden sin recursos (consulta "Escribir una cola productor/consumidor" en la página 946).

Retornar valores

Task tiene una subclase genérica llamada Task<TResult>, que permite que una tarea emita un valor de retorno. Puedes obtener un Task<TResult> llamando a Task.Run con un delegado Func<TResult> (o una expresión lambda compatible) en lugar de un Action:

```
Task<int> task = Task.Run (() => { Console.WriteLine ("Foo"); return 3; });
// ...
```

Puedes obtener el resultado más tarde consultando la propiedad Result. Si la tarea aún no ha terminado, acceder a esta propiedad bloqueará el hilo actual hasta que la tarea termine:

```
int result = task.Result;      // Blocks if not already finished
Console.WriteLine (result);    // 3
```

En el siguiente ejemplo, creamos una tarea que utiliza LINQ para contar la cantidad de números primos en los primeros tres millones (+2) de enteros:

```
Task<int> primeNumberTask = Task.Run (() =>
  Enumerable.Range (2, 3000000).Count (n =>
    Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));

Console.WriteLine ("Task running...");
Console.WriteLine ("The answer is " + primeNumberTask.Result);
```

Esto escribe "Tarea en ejecución..." y luego, unos segundos después, muestra el resultado de 216816.



Task<TResult> puede pensarse como un "futuro" en el sentido de que encapsula un Resultado que estará disponible en algún momento posterior.

Excepciones (Exceptions)

A diferencia de los hilos, las tareas propagan las excepciones de manera conveniente. Por lo tanto, si el código en tu tarea lanza una excepción no controlada (es decir, si tu tarea falla), esa excepción se volverá a lanzar automáticamente a quien llame a Wait() o acceda a la propiedad Result de una Task<TResult>:

```
// Start a Task that throws a NullReferenceException:
Task task = Task.Run (() => { throw null; });
try
{
  task.Wait();
}
catch (AggregateException aex)
{
  if (aex.InnerException is NullReferenceException)
    Console.WriteLine ("Null!");
  else
    throw;
}
```

El CLR envuelve la excepción en un AggregateException para funcionar correctamente en escenarios de programación en paralelo. Discutiremos esto en el Capítulo 22.

Puedes comprobar si una tarea ha fallado sin volver a lanzar la excepción utilizando las propiedades IsFaulted y IsCanceled de la tarea. Si ambas propiedades devuelven false, no se produjo ningún error. Si IsCanceled es true, se lanzó una OperationCanceledException para esa tarea (consulta "Cancelación" en la página 918). Si IsFaulted es true, se lanzó otro tipo de excepción, y la propiedad Exception indicará el error.

Excepciones y tareas autónomas

Con las tareas autónomas ("establecer y olvidar"), es una buena práctica manejar explícitamente las excepciones en el código de la tarea para evitar fallos silenciosos, al igual que harías con un hilo.



Ignorar excepciones está bien cuando una excepción indica únicamente un fallo al obtener un resultado en el que ya no estás interesado. Por ejemplo, si un usuario cancela una solicitud para descargar una página web, no nos importaría si resulta que la página web no existía.

Ignorar excepciones es problemático cuando una excepción indica un error en tu programa, por dos razones:

- El error puede haber dejado tu programa en un estado no válido.
- Pueden ocurrir más excepciones posteriormente como resultado del error, y no registrar el error inicial puede dificultar el diagnóstico.

Puedes suscribirte a excepciones no observadas a nivel global a través del evento estático TaskScheduler.UnobservedTaskException. Manejar este evento y registrar el error puede tener mucho sentido.

Existen un par de matices interesantes en lo que se considera "no observado":

- Las tareas esperadas con un tiempo de espera generarán una excepción no observada si el fallo ocurre después del intervalo de tiempo de espera.
- El acto de verificar la propiedad Exception de una tarea después de que haya fallado hace que la excepción sea "observada".

Continuaciones

Una continuación le dice a una tarea: "Cuando hayas terminado, continúa haciendo algo más". Una continuación generalmente se implementa mediante un callback que se ejecuta una vez que se completa una operación. Hay dos formas de adjuntar una continuación a una tarea.

La primera forma es particularmente significativa porque se utiliza en las funciones asíncronas de C#, como verás pronto. Podemos demostrarlo con la tarea de contar números primos que escribimos hace un momento en la sección "Retornando valores" en la página 628:

```
Task<int> primeNumberTask = Task.Run (() =>
  Enumerable.Range (2, 3000000).Count (n =>
    Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));

var awaiter = primeNumberTask.GetAwaiter();
awaiter.OnCompleted (() =>
{
  int result = awaiter.GetResult();
  Console.WriteLine (result);       // Writes result
});
```

Llamar a GetAwaiter en la tarea devuelve un objeto awaiter cuyo método OnCompleted indica a la tarea antecedente (primeNumberTask) que ejecute un delegado cuando termine (o falle). Es válido adjuntar una continuación a una tarea que ya se ha completado, en cuyo caso la continuación se programará para ejecutarse de inmediato.



Un awaiter es cualquier objeto que expone los dos métodos que acabamos de ver (OnCompleted y GetResult) y una propiedad booleana llamada IsCompleted. No existe una interfaz o clase base que unifique todos estos miembros (aunque OnCompleted forma parte de la interfaz INotifyCompletion). Explicamos la importancia de este patrón en la sección "Funciones asíncronas en C#" en la página 639.

Si una tarea antecedente falla, la excepción se vuelve a lanzar cuando el código de continuación llama a awaiter.GetResult(). En lugar de llamar a GetResult, podríamos acceder simplemente a la propiedad Result del antecedente. La ventaja de llamar a GetResult es que si la tarea antecedente falla, la excepción se lanza directamente sin envolverse en AggregateException, lo que permite bloques catch más simples y limpios.

Para tareas no genéricas, GetResult() tiene un valor de retorno void. Su función útil es simplemente relanzar excepciones.

Si un contexto de sincronización está presente, OnCompleted captura automáticamente dicho contexto y envía la continuación a ese contexto. Esto es muy útil en aplicaciones cliente ricas porque devuelve la continuación al subproceso de la interfaz de usuario (UI). Sin embargo, al escribir bibliotecas, generalmente no es deseable porque el costo relativamente alto de volver al subproceso de la interfaz de usuario debería ocurrir solo una vez al salir de la biblioteca en lugar de entre llamadas de método. Por lo tanto, puedes evitarlo utilizando el método ConfigureAwait:

```
var awaiter = primeNumberTask.ConfigureAwait (false).GetAwaiter();
```

Si no hay un contexto de sincronización presente, o si usas ConfigureAwait(false), la continuación se ejecutará (en general) en el mismo subproceso que el antecedente, evitando gastos innecesarios.

La otra forma de adjuntar una continuación es llamando al método ContinueWith de la tarea:

```
primeNumberTask.ContinueWith (antecedent =>
{
  int result = antecedent.Result;
  Console.WriteLine (result);          // Writes 123
});
```

ContinueWith devuelve una tarea, lo cual es útil si deseas adjuntar más continuaciones. Sin embargo, debes lidiar directamente con AggregateException si la tarea falla y escribir código adicional para enviar la continuación en aplicaciones de UI (consultar "Planificadores de tareas" en la página 939). En contextos que no son de UI, debes especificar TaskContinuationOptions.ExecuteSynchronously si deseas que la continuación se ejecute en el mismo subproceso; de lo contrario, se enviará al grupo de subprocesos. ContinueWith es especialmente útil en escenarios de programación paralela, los cuales se cubren en detalle en el Capítulo 22.

TaskCompletionSource

Hemos visto cómo Task.Run crea una tarea que ejecuta un delegado en un subproceso agrupado (o no agrupado). Otra forma de crear una tarea es con TaskCompletionSource. TaskCompletionSource te permite crear una tarea a partir de cualquier operación que se complete en el futuro. Funciona mediante la creación de una tarea "esclava" que controlas manualmente, indicando cuándo finaliza o falla la operación. Esto es ideal para trabajos basados en E/S: obtienes todos los beneficios de las tareas (con su capacidad para propagar valores de retorno, excepciones y continuaciones) sin bloquear un subproceso durante la duración de la operación.

Para usar TaskCompletionSource, simplemente instancias la clase. Expone una propiedad Task que devuelve una tarea en la cual puedes esperar y adjuntar continuaciones, al igual que con cualquier otra tarea. Sin embargo, la tarea es controlada completamente por el objeto TaskCompletionSource a través de los siguientes métodos:

```
public class TaskCompletionSource<TResult>
{
  public void SetResult (TResult result);

   public void SetException (Exception exception);
   public void SetCanceled();

   public bool TrySetResult (TResult result);
   public bool TrySetException (Exception exception);
   public bool TrySetCanceled();
   public bool TrySetCanceled (CancellationToken cancellationToken);
   ...
 }
```

Llamar a cualquiera de estos métodos indica a la tarea, colocándola en un estado completado, con error o cancelado (este último se cubre en la sección "Cancelación" en la página 659). Se supone que debes llamar a uno de estos métodos exactamente una vez: si se llama nuevamente, SetResult, SetException o SetCanceled lanzarán una excepción, mientras que los métodos Try* devolverán false.

El siguiente ejemplo imprime 42 después de esperar cinco segundos:

```
var tcs = new TaskCompletionSource<int>();

new Thread (() => { Thread.Sleep (5000); tcs.SetResult (42); })
  { IsBackground = true }
  .Start();

Task<int> task = tcs.Task;         // Our "slave" task.
Console.WriteLine (task.Result);   // 42
```

Con TaskCompletionSource, podemos escribir nuestro propio método Run:

```
Task<TResult> Run<TResult> (Func<TResult> function)
{
  var tcs = new TaskCompletionSource<TResult>();
  new Thread (() =>
  {
    try { tcs.SetResult (function()); }
    catch (Exception ex) { tcs.SetException (ex); }
  }).Start();
  return tcs.Task;
}
...
Task<int> task = Run (() => { Thread.Sleep (5000); return 42; });
```

Llamar a este método es equivalente a llamar a Task.Factory.StartNew con la opción de creación de tareas TaskCreationOptions.LongRunning para solicitar un subproceso no agrupado.

El verdadero poder de TaskCompletionSource radica en la creación de tareas que no bloquean subprocesos. Por ejemplo, considera una tarea que espera cinco segundos y luego devuelve el número 42. Podemos escribir esto sin un subproceso utilizando la clase Timer, que, con la ayuda del CLR (y a su vez, el sistema operativo), dispara un evento en x milisegundos (volveremos a los temporizadores en el Capítulo 21):

```
Task<int> GetAnswerToLife()
{
  var tcs = new TaskCompletionSource<int>();
  // Create a timer that fires once in 5000 ms:

   var timer = new System.Timers.Timer (5000) { AutoReset = false };
   timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult (42); };
   timer.Start();
   return tcs.Task;
 }
```

Por lo tanto, nuestro método devuelve una tarea que se completa cinco segundos después, con un resultado de 42. Al adjuntar una continuación a la tarea, podemos escribir su resultado sin bloquear ningún subproceso:

```
var awaiter = GetAnswerToLife().GetAwaiter();
awaiter.OnCompleted (() => Console.WriteLine (awaiter.GetResult()));
```

Podríamos hacer esto más útil y convertirlo en un método de retraso de propósito general al parametrizar el tiempo de retraso y eliminar el valor de retorno. Esto significa que debe devolver una tarea en lugar de una Task<int>. Sin embargo, no hay una versión no genérica de TaskCompletionSource, lo que significa que no podemos crear directamente una Task no genérica. La solución alternativa es simple: debido a que Task<TResult> deriva de Task, creamos un TaskCompletionSource<anything> y luego convertimos implícitamente el Task<anything> que nos da en un Task, de esta manera:

```
var tcs = new TaskCompletionSource<object>();
Task task = tcs.Task;
```

Ahora podemos escribir nuestro método de retraso de propósito general:

```
Task Delay (int milliseconds)
{
  var tcs = new TaskCompletionSource<object>();
  var timer = new System.Timers.Timer (milliseconds) { AutoReset = false };
  timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult (null); };
  timer.Start();
  return tcs.Task;
}
```



.NET 5 introduce un TaskCompletionSource no genérico, por lo que si estás utilizando .NET 5 o superior, puedes sustituir TaskCompletionSource<object> por TaskCompletionSource.

Así es cómo podemos usarlo para escribir "42" después de cinco segundos:

```
Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));
```

Nuestro uso de TaskCompletionSource sin un hilo significa que un hilo se involucra solo cuando comienza la continuación, cinco segundos después. Podemos demostrar esto iniciando 10,000 de estas operaciones al mismo tiempo sin errores o consumo excesivo de recursos:

```
for (int i = 0; i < 10000; i++)
  Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));
```



Los temporizadores disparan sus devoluciones de llamada en subprocesos del grupo de subprocesos, por lo que después de cinco segundos, el grupo de subprocesos recibirá 10,000 solicitudes para llamar a SetResult(null) en un TaskCompletionSource. Si las solicitudes llegan más rápido de lo que se pueden procesar, el grupo de subprocesos responderá encolándolas y luego procesándolas al nivel óptimo de paralelismo para la CPU. Esto es ideal si los trabajos enlazados al subproceso son de corta duración, lo cual es cierto en este caso: el trabajo enlazado al subproceso es simplemente la llamada a SetResult más la acción de enviar la continuación al contexto de sincronización (en una aplicación de UI) o, de lo contrario, la propia continuación (Console.WriteLine(42)).

Task.Delay

El método Delay que acabamos de escribir es lo suficientemente útil como para estar disponible como un método estático en la clase Task:

```
Task.Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));
```

o:

```
Task.Delay (5000).ContinueWith (ant => Console.WriteLine (42));
```

Task.Delay es el equivalente asíncrono de Thread.Sleep.

Principios de la Asincronía

Al demostrar TaskCompletionSource, terminamos escribiendo métodos asíncronos. En esta sección, definimos exactamente qué son las operaciones asíncronas y explicamos cómo esto conduce a la programación asíncrona.

Operaciones Síncronas versus Operaciones Asíncronas

Una operación síncrona realiza su trabajo antes de regresar al llamador.

Una operación asíncrona puede hacer (la mayor parte o todo) su trabajo después de regresar al llamador.

La mayoría de los métodos que escribes y llamas son síncronos. Un ejemplo es List<T>.Add, o Console.WriteLine, o Thread.Sleep. Los métodos asíncronos son menos comunes e inician la concurrencia, porque el trabajo continúa en paralelo al llamador. Los métodos asíncronos suelen regresar rápidamente (o inmediatamente) al llamador; por lo tanto, también se les llama métodos sin bloqueo.

La mayoría de los métodos asíncronos que hemos visto hasta ahora se pueden describir como métodos de propósito general:

- Thread.Start
- Task.Run
- Métodos que adjuntan continuaciones a tareas

Además, algunos de los métodos que discutimos en "Contextos de Sincronización" en la página 623 (Dispatcher.BeginInvoke, Control.BeginInvoke y SynchronizationContext.Post) son asíncronos, al igual que los métodos que escribimos en "TaskCompletionSource" en la página 631, incluyendo Delay.

¿Qué es la Programación Asíncrona?

El principio de la programación asíncrona es que escribas funciones de larga duración (o potencialmente larga duración) de manera asíncrona. Esto contrasta con el enfoque convencional de escribir funciones de larga duración de manera sincrónica y luego llamar esas funciones desde un nuevo hilo o tarea para introducir concurrencia según sea necesario.

La diferencia con el enfoque asíncrono es que la concurrencia se inicia dentro de la función de larga duración en lugar de desde fuera de la función. Esto tiene dos beneficios:

- La concurrencia vinculada a operaciones de E/S se puede implementar sin bloquear hilos (como demostramos en "TaskCompletionSource" en la página 631), lo que mejora la escalabilidad y eficiencia.
- Las aplicaciones de cliente rico terminan con menos código en hilos de trabajo, simplificando la seguridad de hilos.

Esto, a su vez, conduce a dos usos distintos para la programación asíncrona. El primero es escribir aplicaciones (por lo general, del lado del servidor) que manejan eficientemente una gran cantidad de E/S concurrente. El desafío aquí no es la seguridad de hilos (porque generalmente

hay un mínimo de estado compartido), sino la eficiencia de hilos; en particular, no consumir un hilo por solicitud de red. Por lo tanto, en este contexto, solo las operaciones vinculadas a E/S se benefician de la asincronía.

El segundo uso es simplificar la seguridad de hilos en aplicaciones de cliente rico. Esto es especialmente relevante a medida que un programa crece en tamaño, porque para lidiar con la complejidad, generalmente refactorizamos métodos más grandes en métodos más pequeños, lo que resulta en cadenas de métodos que se llaman entre sí (gráficos de llamadas).

Con un gráfico de llamadas sincrónico tradicional, si alguna operación dentro del gráfico de llamadas tiene una larga duración, debemos ejecutar todo el gráfico de llamadas en un hilo de trabajo para mantener una interfaz de usuario receptiva. Por lo tanto, terminamos con una única operación concurrente que abarca muchos métodos (concurrencia a gran escala), lo que requiere considerar la seguridad de hilos para cada método en el gráfico.

Con un gráfico de llamadas asíncrono, no necesitamos iniciar un hilo hasta que realmente sea necesario, generalmente al final del gráfico (o incluso no en el caso de operaciones vinculadas a E/S). Todos los demás métodos pueden ejecutarse completamente en el hilo de la interfaz de usuario, con una seguridad de hilos mucho más simple. Esto resulta en una concurrencia de granularidad fina: una secuencia de pequeñas operaciones concurrentes, entre las cuales la ejecución regresa al hilo de la interfaz de usuario.



Para aprovechar esto, tanto las operaciones vinculadas a E/S como las operaciones vinculadas a cómputo deben escribirse de manera asíncrona; una buena regla general es incluir cualquier cosa que pueda tomar más de 50 ms. (Por otro lado, una asincronía excesivamente detallada puede afectar el rendimiento, porque las operaciones asíncronas incurren en una sobrecarga, consulta "Optimizaciones" en la página 655).

En este capítulo, nos centramos principalmente en el escenario de aplicaciones de cliente rico, que es el más complejo de los dos. En el Capítulo 16, presentamos dos ejemplos que ilustran el escenario vinculado a E/S (consulta "Concurrencia con TCP" en la página 738 y "Escribir un servidor HTTP" en la página 731).



El marco de trabajo UWP fomenta la programación asíncrona hasta el punto en que las versiones síncronas de algunos métodos de larga duración no se exponen o generan excepciones. En su lugar, debes llamar a métodos asíncronos que devuelven tareas (o objetos que se pueden convertir en tareas mediante el método de extensión AsTask).

Programación Asíncrona y Continuaciones

Las tareas son ideales para la programación asíncrona porque admiten continuaciones, que son esenciales para la asincronía (considera el método Delay que escribimos en "TaskCompletionSource" en la página 631). Al escribir Delay, utilizamos TaskCompletionSource, que es una forma estándar de implementar métodos asíncronos de bajo nivel vinculados a E/S.

Para los métodos vinculados a cómputo, utilizamos Task.Run para iniciar la concurrencia vinculada a hilos. Simplemente al devolver la tarea al llamador, creamos un método asíncrono.

Lo que distingue a la programación asíncrona es que nuestro objetivo es hacerlo en un nivel inferior del gráfico de llamadas para que, en aplicaciones de cliente rico, los métodos de nivel superior puedan permanecer en el hilo de la interfaz de usuario y acceder a controles y estado compartido sin problemas de seguridad de hilos. Para ilustrar esto, considera el siguiente método que calcula y cuenta números primos utilizando todos los núcleos disponibles (hablaremos de ParallelEnumerable en el Capítulo 22):

```
int GetPrimesCount (int start, int count)
{
  return
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0));
}
```

Los detalles de cómo funciona esto no son importantes; lo que importa es que puede llevar algún tiempo ejecutarse. Podemos demostrar esto escribiendo otro método para llamarlo:

```
void DisplayPrimeCounts()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine (GetPrimesCount (i*1000000 + 2, 1000000) +
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1));
  Console.WriteLine ("Done!");
}
```

Aquí esta la salida:

```
78498 primes between 0 and 999999
70435 primes between 1000000 and 1999999
67883 primes between 2000000 and 2999999
66330 primes between 3000000 and 3999999
65367 primes between 4000000 and 4999999
64336 primes between 5000000 and 5999999
63799 primes between 6000000 and 6999999
63129 primes between 7000000 and 7999999
62712 primes between 8000000 and 8999999
62090 primes between 9000000 and 9999999
```

Ahora tenemos un gráfico de llamadas, con DisplayPrimeCounts llamando a GetPrimesCount. El primero utiliza Console.WriteLine por simplicidad, aunque en realidad es más probable que esté actualizando controles de interfaz de usuario en una aplicación de cliente rico, como demostraremos más adelante.

Podemos iniciar la concurrencia a nivel grueso para este gráfico de llamadas de la siguiente manera:

```
Task.Run (() => DisplayPrimeCounts());
```

Con un enfoque asíncrono a nivel fino, en cambio, comenzamos escribiendo una versión asíncrona de GetPrimesCount:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
  return Task.Run (() =>
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

Por qué es importante el soporte del lenguaje

Ahora debemos modificar DisplayPrimeCounts para que llame a GetPrimesCountAsync. Aquí es donde entran en juego las palabras clave await y async de C#, porque hacerlo de otra manera es más complicado de lo que parece. Si simplemente modificamos el bucle de la siguiente manera:

```
for (int i = 0; i < 10; i++)
{
  var awaiter = GetPrimesCountAsync (i*1000000 + 2, 1000000).GetAwaiter();
  awaiter.OnCompleted (() =>
    Console.WriteLine (awaiter.GetResult() + " primes between... "));
}
Console.WriteLine ("Done");
```

el bucle girará rápidamente a través de 10 iteraciones (los métodos son no bloqueantes) y las 10 operaciones se ejecutarán en paralelo (seguidas de un "Done" prematuro).



Ejecutar estas tareas en paralelo no es deseable en este caso porque sus implementaciones internas ya están paralelizadas; solo hará que esperemos más tiempo para ver los primeros resultados (y desordenará el orden).

Sin embargo, hay una razón mucho más común para necesitar serializar la ejecución de tareas, que es que la Tarea B depende del resultado de la Tarea A. Por ejemplo, al obtener una página web, una consulta DNS debe preceder a la solicitud HTTP.

Para ejecutarlos de forma secuencial, debemos desencadenar la próxima iteración del bucle desde la continuación misma. Esto significa eliminar el bucle for y recurrir a una llamada recursiva en la continuación:

```
void DisplayPrimeCounts()
{
  DisplayPrimeCountsFrom (0);
}

void DisplayPrimeCountsFrom (int i)
{
  var awaiter = GetPrimesCountAsync (i*1000000 + 2, 1000000).GetAwaiter();
  awaiter.OnCompleted (() =>
  {
    Console.WriteLine (awaiter.GetResult() + " primes between...");
    if (++i < 10) DisplayPrimeCountsFrom (i);
    else Console.WriteLine ("Done");
  });
}
```

La situación empeora si queremos hacer que DisplayPrimeCounts en sí sea asíncrono, devolviendo una tarea que se señala al completarse. Para lograr esto, es necesario crear un TaskCompletionSource:

```
Task DisplayPrimeCountsAsync()
{
  var machine = new PrimesStateMachine();
  machine.DisplayPrimeCountsFrom (0);
  return machine.Task;
}

class PrimesStateMachine
{
  TaskCompletionSource<object> _tcs = new TaskCompletionSource<object>();
  public Task Task { get { return _tcs.Task; } }

  public void DisplayPrimeCountsFrom (int i)
  {
    var awaiter = GetPrimesCountAsync (i*1000000+2, 1000000).GetAwaiter();
    awaiter.OnCompleted (() =>
    {
      Console.WriteLine (awaiter.GetResult());
      if (++i < 10) DisplayPrimeCountsFrom (i);
      else { Console.WriteLine ("Done"); _tcs.SetResult (null); }
    });

  }
}
```

Afortunadamente, las funciones asíncronas de C# hacen todo este trabajo por nosotros. Con las palabras clave async y await, solo necesitamos escribir esto:

```
async Task DisplayPrimeCountsAsync()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine (await GetPrimesCountAsync (i*1000000 + 2, 1000000) +
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1));
  Console.WriteLine ("Done!");
}
```

En consecuencia, async y await son esenciales para implementar la asincronía sin una complejidad excesiva. Veamos ahora cómo funcionan estas palabras clave.



Otra forma de ver el problema es que las construcciones de bucles imperativos (for, foreach, etc.) no se mezclan bien con las continuaciones, porque dependen del estado local actual del método ("¿Cuántas veces más se ejecutará este bucle?").

Aunque las palabras clave async y await ofrecen una solución, a veces es posible resolverlo de otra manera reemplazando las construcciones de bucles imperativos por el equivalente funcional (es decir, consultas LINQ). Esto es la base de las Extensiones Reactivas (Rx) y puede ser una buena opción cuando se desea ejecutar operadores de consulta sobre el resultado o combinar múltiples secuencias. El precio a pagar es que, para evitar el bloqueo, Rx opera sobre secuencias basadas en empuje (push), lo que puede ser conceptualmente complicado.

Funciones Asíncronas en C#

Las palabras clave async y await te permiten escribir código asíncrono que tiene la misma estructura y simplicidad que el código síncrono, al tiempo que elimina la "plomería" de la programación asíncrona.

Awaiting (Esperando)

La palabra clave await simplifica la adhesión de continuaciones. Comenzando con un escenario básico, el compilador expande esto:

```
var result = await expression;
statement(s);
```

en algo funcionalmente similar a esto:

```
var awaiter = expression.GetAwaiter();
awaiter.OnCompleted (() =>
{
  var result = awaiter.GetResult();

  statement(s);
});
```



El compilador también genera código para omitir la continuación en caso de finalización síncrona (ver "Optimizaciones" en la página 655) y para manejar varias sutilezas que veremos en secciones posteriores.

Para demostrarlo, volvamos a visitar el método asíncrono que escribimos anteriormente que calcula y cuenta números primos:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
  return Task.Run (() =>
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

Con la palabra clave `await`, podemos llamarlo de la siguiente manera:

```
int result = await GetPrimesCountAsync (2, 1000000);
Console.WriteLine (result);
```

Para compilar, necesitamos agregar el modificador `async` al método contenedor:

```
async void DisplayPrimesCount()
{
  int result = await GetPrimesCountAsync (2, 1000000);
  Console.WriteLine (result);
}
```

El modificador `async` le indica al compilador que trate `await` como una palabra clave en lugar de un identificador en caso de que surja una ambigüedad dentro de ese método (esto asegura que el código escrito antes de C# 5 que podría usar `await` como un identificador aún se compile sin errores). El modificador `async` solo se puede aplicar a métodos (y expresiones lambda) que devuelven `void` o (como verás más adelante) un `Task` o `Task<TResult>`.



El modificador `async` es similar al modificador `unsafe` en el sentido de que no tiene efecto en la firma del método o en los metadatos públicos; solo afecta lo que sucede dentro del método. Por esta razón, no tiene sentido usar `async` en una interfaz. Sin embargo, es válido, por ejemplo, introducir `async` al anular un método virtual no `async`, siempre y cuando mantengas la misma firma.

Los métodos con el modificador `async` se llaman funciones asíncronas, porque ellos mismos suelen ser asíncronos. Para entender por qué, veamos cómo se ejecuta a través de una función asíncrona.

Al encontrarse con una expresión await, la ejecución (normalmente) regresa al llamador, de manera similar a como ocurre con yield return en un iterador. Sin embargo, antes de regresar, el tiempo de ejecución adjunta una continuación a la tarea esperada, asegurando que cuando la tarea se complete, la ejecución regrese al método y continúe desde donde se detuvo. Si la tarea falla, se vuelve a lanzar su excepción, de lo contrario, su valor de retorno se asigna a la expresión await.

Podemos resumir todo lo que hemos dicho hasta ahora observando la expansión lógica del método asíncrono que acabamos de examinar:

```
void DisplayPrimesCount()
{
  var awaiter = GetPrimesCountAsync (2, 1000000).GetAwaiter();
  awaiter.OnCompleted (() =>
  {
    int result = awaiter.GetResult();
    Console.WriteLine (result);
  });
}
```

La expresión en la que se realiza el await normalmente es una tarea (Task), sin embargo, cualquier objeto con un método GetAwaiter que devuelva un awaiter (que implemente INotifyCompletion.OnCompleted y tenga un método GetResult con el tipo apropiado y una propiedad IsCompleted de tipo bool) será aceptado por el compilador.

Observa que nuestra expresión await se evalúa como un tipo int; esto se debe a que la expresión que esperamos era un Task<int> (cuyo método GetAwaiter().GetResult() devuelve un int).

Es válido esperar una tarea no genérica, lo cual generará una expresión void:

```
await Task.Delay (5000);
Console.WriteLine ("Five seconds passed!");
```

Capturando estado local

El verdadero poder de las expresiones await es que pueden aparecer casi en cualquier lugar del código. Específicamente, una expresión await puede aparecer en lugar de cualquier expresión (dentro de una función asíncrona), excepto dentro de una expresión lock o un contexto unsafe.

En el siguiente ejemplo, usamos await dentro de un bucle:

```
async void DisplayPrimeCounts()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine (await GetPrimesCountAsync (i*1000000+2, 1000000));
}
```

Al ejecutar GetPrimesCountAsync por primera vez, la ejecución regresa al llamador gracias a la expresión await. Cuando el método se completa (o falla), la ejecución se reanuda desde donde se detuvo, preservando los valores de las variables locales y los contadores del bucle.

Sin la palabra clave await, la forma más simple equivalente podría ser el ejemplo que escribimos en "Why Language Support Is Important" en la página 637. Sin embargo, el compilador adopta una estrategia más general al refactorizar tales métodos en máquinas de estado (similar a lo que hace con los iteradores).

El compilador confía en continuaciones (mediante el patrón awaiter) para reanudar la ejecución después de una expresión await. Esto significa que, si se está ejecutando en el hilo de interfaz de usuario (UI) de una aplicación de cliente, el contexto de sincronización asegura que la ejecución se reanude en el mismo hilo. De lo contrario, la ejecución se reanuda en el hilo en el que la tarea finalizó. El cambio de hilo no afecta el orden de ejecución y tiene poca importancia a menos que dependas de la afinidad de hilos, quizás a través del uso de almacenamiento local de hilos (ver "Thread-Local Storage" en la página 898). Es como recorrer una ciudad y llamar taxis para ir de un destino a otro. Con un contexto de sincronización, siempre obtendrás el mismo taxi; sin un contexto de sincronización, generalmente obtendrás un taxi diferente cada vez. En cualquier caso, el viaje es el mismo.

Esperando en una interfaz de usuario (UI)

Podemos demostrar las funciones asíncronas en un contexto más práctico al escribir una interfaz de usuario simple que permanezca receptiva mientras llama a un método que consume tiempo de cálculo. Comencemos con una solución sincrónica:

```
class TestUI : Window
{
  Button _button = new Button { Content = "Go" };
  TextBlock _results = new TextBlock();

  public TestUI()
  {
    var panel = new StackPanel();
    panel.Children.Add (_button);
    panel.Children.Add (_results);
    Content = panel;
    _button.Click += (sender, args) => Go();
  }

  void Go()
  {
    for (int i = 1; i < 5; i++)
      _results.Text += GetPrimesCount (i * 1000000, 1000000) +
        " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1) +
        Environment.NewLine;
  }

  int GetPrimesCount (int start, int count)
  {
    return ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0));
  }
}
```

Al presionar el botón "Go", la aplicación se vuelve no receptiva durante el tiempo que tarda en ejecutarse el código que consume tiempo de cálculo. Hay dos pasos para asincronizar esto; el primero es cambiar a la versión asíncrona de GetPrimesCount que utilizamos en ejemplos anteriores:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
  return Task.Run (() =>
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

El segundo paso es modificar el método "Go" para llamar a "GetPrimesCountAsync":

```
async void Go()
{
  _button.IsEnabled = false;
  for (int i = 1; i < 5; i++)
    _results.Text += await GetPrimesCountAsync (i * 1000000, 1000000) +
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1) +
      Environment.NewLine;
  _button.IsEnabled = true;
}
```

Esto ilustra la simplicidad de programar con funciones asíncronas: programas como lo harías sincrónicamente, pero llamas a funciones asíncronas en lugar de funciones bloqueantes y las esperas. Solo el código dentro de "GetPrimesCountAsync" se ejecuta en un hilo de trabajo,

mientras que el código en "Go" "arrienda" tiempo en el hilo de la interfaz de usuario. Podríamos decir que "Go" se ejecuta de manera pseudoconcurrente al bucle de mensajes (en el sentido de que su ejecución se intercala con otros eventos que el hilo de la interfaz de usuario procesa). Con esta pseudoconcurrencia, el único punto en el que puede ocurrir una preempción es durante un await. Esto simplifica la seguridad de subprocesos: en nuestro caso, el único problema que esto podría causar es la reentrancia (hacer clic en el botón nuevamente mientras se está ejecutando, lo cual evitamos deshabilitando el botón). La verdadera concurrencia ocurre más abajo en la pila de llamadas, dentro del código llamado por Task.Run. Para beneficiarse de este modelo, el código verdaderamente concurrente evita acceder a estados compartidos o controles de la interfaz de usuario.

Para dar otro ejemplo, supongamos que en lugar de calcular números primos, queremos descargar varias páginas web y sumar sus longitudes. .NET expone numerosos métodos asíncronos que devuelven una tarea, uno de los cuales es la clase WebClient en System.Net. El método DownloadDataTaskAsync descarga de forma asíncrona un URI a una matriz de bytes, devolviendo una Task<byte[]>, por lo que al esperarla, obtenemos un byte[]. Ahora vamos a reescribir nuestro método "Go":

```
async void Go()
{
  _button.IsEnabled = false;
  string[] urls = "www.albahari.com www.oreilly.com www.linqpad.net".Split();
  int totalLength = 0;
  try
  {
    foreach (string url in urls)
    {
      var uri = new Uri ("http://" + url);
      byte[] data = await new WebClient().DownloadDataTaskAsync (uri);
      _results.Text += "Length of " + url + " is " + data.Length +
                       Environment.NewLine;
      totalLength += data.Length;


    }
    _results.Text += "Total length: " + totalLength;
  }
  catch (WebException ex)
  {
    _results.Text += "Error: " + ex.Message;
  }
  finally { _button.IsEnabled = true; }
}
```

Una vez más, esto refleja cómo lo escribiríamos de manera sincrónica, incluyendo el uso de bloques catch y finally. Aunque la ejecución regresa al llamador después del primer await, el bloque finally no se ejecuta hasta que el método haya completado lógicamente (por virtud de que todo su código se haya ejecutado, o por un retorno anticipado o una excepción no controlada).

Puede ser útil considerar exactamente lo que está sucediendo en el fondo. Primero, debemos revisar el pseudocódigo que ejecuta el bucle de mensajes en el hilo de la interfaz de usuario:

```
Set synchronization context for this thread to WPF sync context
while (!thisApplication.Ended)
{
  wait for something to appear in message queue
  Got something: what kind of message is it?
    Keyboard/mouse message -> fire an event handler
    User BeginInvoke/Invoke message -> execute delegate
}
```

Los controladores de eventos que adjuntamos a elementos de la interfaz de usuario se ejecutan a través de este bucle de mensajes. Cuando se ejecuta nuestro método "Go", la ejecución continúa hasta la expresión await y luego regresa al bucle de mensajes (permitiendo que la interfaz de usuario responda a otros eventos). Sin embargo, la expansión del compilador de await asegura que antes de regresar, se establezca una continuación para que la ejecución se reanude donde se dejó una vez que se complete la tarea. Y debido a que esperamos en un hilo de la interfaz de usuario, la continuación se envía al contexto de sincronización, que la ejecuta a través del bucle de mensajes, manteniendo todo nuestro método "Go" ejecutándose pseudo-concurrentemente en el hilo de la interfaz de usuario. La verdadera concurrencia (vinculada a operaciones de E/S) ocurre dentro de la implementación de DownloadDataTaskAsync.

Comparación con la concurrencia de granularidad gruesa

La programación asíncrona era difícil antes de C# 5, no solo porque no había soporte del lenguaje, sino porque el .NET Framework exponía funcionalidad asíncrona a través de patrones torpes llamados EAP y APM (ver "Patrones Obsoletos" en la página 667) en lugar de métodos que devolvieran tareas.

La solución popular era la concurrencia de granularidad gruesa (de hecho, incluso había un tipo llamado BackgroundWorker para ayudar con eso). Volviendo a nuestro ejemplo original sincrónico con GetPrimesCount, podemos demostrar la asincronía de granularidad gruesa modificando el controlador de eventos del botón de la siguiente manera:

```
...
_button.Click += (sender, args) =>
{


  _button.IsEnabled = false;
  Task.Run (() => Go());
};
```

(Hemos elegido usar Task.Run en lugar de BackgroundWorker porque este último no simplificaría nuestro ejemplo en particular). En ambos casos, el resultado final es que todo nuestro gráfico de llamadas sincrónicas (Go más GetPrimesCount) se ejecuta en un hilo de trabajo. Y debido a que Go actualiza elementos de la interfaz de usuario, ahora debemos esparcir nuestro código con Dispatcher.BeginInvoke:

```
void Go()
{
  for (int i = 1; i < 5; i++)
  {
    int result = GetPrimesCount (i * 1000000, 1000000);
    Dispatcher.BeginInvoke (new Action (() =>
      _results.Text += result + " primes between " + (i*1000000) +
      " and " + ((i+1)*1000000-1) + Environment.NewLine));
  }
  Dispatcher.BeginInvoke (new Action (() => _button.IsEnabled = true));
}
```

A diferencia de la versión asíncrona, el bucle en sí se ejecuta en un hilo de trabajo. Esto puede parecer inofensivo, sin embargo, incluso en este caso simple, nuestro uso de múltiples hilos ha introducido una condición de carrera (¿puedes identificarla? Si no puedes, intenta ejecutar el programa: casi con seguridad se hará evidente).

Implementar la cancelación y el informe de progreso crea más posibilidades de errores de seguridad de subprocesos, al igual que cualquier código adicional en el método. Por ejemplo, supongamos que el límite superior para el bucle no está codificado de forma rígida, sino que proviene de una llamada a un método:

```
for (int i = 1; i < GetUpperBound(); i++)
```

Ahora supongamos que GetUpperBound() lee el valor de un archivo de configuración cargado de forma perezosa, que se carga desde el disco en la primera llamada. Todo este código ahora se ejecuta en tu hilo de trabajo, y es muy probable que no sea seguro en términos de subprocesos. Este es el peligro de comenzar hilos de trabajo en lo alto del gráfico de llamadas.

Escribir Funciones Asíncronas

Con cualquier función asíncrona, puedes reemplazar el tipo de retorno void con Task para que el método en sí sea útilmente asíncrono (y esperable). No se requieren más cambios:

```
async Task PrintAnswerToLife()   // We can return Task instead of void
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  Console.WriteLine (answer);
}
```

Observa que no retornamos explícitamente una tarea en el cuerpo del método. El compilador fabrica la tarea, que se señaliza al completarse el método (o al producirse una excepción no controlada). Esto facilita la creación de cadenas de llamadas asíncronas:

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Done");
}
```

Y debido a que hemos declarado que Go tiene un tipo de retorno de Task, Go en sí mismo es esperable. El compilador expande las funciones asíncronas que devuelven tareas en código que utiliza TaskCompletionSource para crear una tarea que luego se señaliza o falla.

Dejando de lado los matices, podemos expandir PrintAnswerToLife en el siguiente equivalente funcional:

```
Task PrintAnswerToLife()
{
  var tcs = new TaskCompletionSource<object>();
  var awaiter = Task.Delay (5000).GetAwaiter();
  awaiter.OnCompleted (() =>
  {
    try
    {
      awaiter.GetResult();    // Re-throw any exceptions
      int answer = 21 * 2;
      Console.WriteLine (answer);
      tcs.SetResult (null);
    }
    catch (Exception ex) { tcs.SetException (ex); }
  });
  return tcs.Task;
}
```

Por lo tanto, cada vez que un método asíncrono que devuelve una tarea finaliza, la ejecución vuelve al punto en el que se le hizo la espera (por virtud de una continuación).



En un escenario de cliente rico(rich-client), en este punto la ejecución vuelve al hilo de la interfaz de usuario (si aún no está en él). De lo contrario, continúa en el mismo hilo en el que volvió la continuación. Esto significa que no hay un costo de latencia al propagar gráficos de llamadas asíncronas, aparte del primer "rebote" si se inició en el hilo de la interfaz de usuario.

Retorno de Task<TResult>

Puedes devolver una tarea de tipo Task<TResult> si el cuerpo del método devuelve un TResult:

```
async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;

  return answer;    // Method has return type Task<int> we return int
}
```

Internamente, esto resulta en que TaskCompletionSource se señalice con un valor en lugar de null. Podemos demostrar GetAnswerToLife llamándolo desde PrintAnswerToLife (que a su vez se llama desde Go):

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Done");
}

async Task PrintAnswerToLife()
{
  int answer = await GetAnswerToLife();
  Console.WriteLine (answer);
}

async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  return answer;
}
```

De hecho, hemos refactorizado nuestro original PrintAnswerToLife en dos métodos, con la misma facilidad que si estuviéramos programando de manera síncrona. La similitud con la programación síncrona es intencional; aquí está el equivalente síncrono de nuestro grafo de llamadas, para el cual llamar a Go() da el mismo resultado después de bloquear durante cinco segundos:

```
void Go()
{
  PrintAnswerToLife();
  Console.WriteLine ("Done");
}

void PrintAnswerToLife()
{
  int answer = GetAnswerToLife();
  Console.WriteLine (answer);
}

int GetAnswerToLife()
{
  Thread.Sleep (5000);
  int answer = 21 * 2;
  return answer;
}
```



Esto también ilustra el principio básico de cómo diseñar con funciones asíncronas en C#:

1. Escribe tus métodos de forma síncrona.

2. Reemplaza las llamadas a métodos síncronos por llamadas a métodos asíncronos y espera su resultado.

3. A excepción de los métodos "de nivel superior" (normalmente controladores de eventos para controles de interfaz de usuario), actualiza los tipos de retorno de tus métodos asíncronos a Task o Task<TResult> para que puedan ser esperados.

La capacidad del compilador de crear tareas para funciones asíncronas significa que en su mayoría no necesitas instanciar explícitamente un TaskCompletionSource, excepto en casos relativamente raros en los que se inicia concurrencia basada en E/S en métodos de nivel inferior. (Y para los métodos que inician concurrencia basada en cálculos, creas la tarea con Task.Run).

Ejecución del grafo de llamadas asíncronas

Para comprender exactamente cómo se ejecuta esto, es útil reorganizar nuestro código de la siguiente manera:

```
async Task Go()
{
  var task = PrintAnswerToLife();
  await task; Console.WriteLine ("Done");
}

async Task PrintAnswerToLife()
{
  var task = GetAnswerToLife();
  int answer = await task; Console.WriteLine (answer);
}

async Task<int> GetAnswerToLife()
{
  var task = Task.Delay (5000);
  await task; int answer = 21 * 2; return answer;
}
```

Go llama a PrintAnswerToLife, que llama a GetAnswerToLife, que llama a Delay y luego espera. La espera hace que la ejecución regrese a PrintAnswerToLife, que también espera, y luego regresa a Go, que también espera y devuelve al llamador. Todo esto sucede de forma síncrona, en el hilo que llamó a Go; esta es la breve fase síncrona de ejecución.

Cinco segundos después, se activa la continuación en Delay y la ejecución regresa a GetAnswerToLife en un hilo del grupo de hilos (si comenzamos en un hilo de interfaz de usuario, la ejecución vuelve a ese hilo). A continuación, se ejecutan las declaraciones restantes en GetAnswerToLife y, después de eso, la tarea Task<int> del método se completa con un resultado de 42 y ejecuta la continuación en PrintAnswerToLife, que ejecuta las declaraciones restantes en ese método. El proceso continúa hasta que la tarea de Go se marca como completada.

El flujo de ejecución coincide con el grafo de llamadas síncronas que mostramos anteriormente porque seguimos un patrón en el que esperamos inmediatamente cada método asíncrono después de llamarlo. Esto crea un flujo secuencial sin paralelismo ni ejecución superpuesta dentro del grafo de llamadas. Cada expresión await crea un "hueco" en la ejecución, después del cual el programa continúa donde lo dejó.

Paralelismo

Llamar a un método asíncrono sin esperarlo permite que el código que sigue se ejecute en paralelo. Es posible que hayas notado en ejemplos anteriores que teníamos un botón cuyo controlador de eventos llamaba a Go de la siguiente manera:

```
_button.Click += (sender, args) => Go();
```

A pesar de que Go es un método asíncrono, no lo esperamos, y esto es precisamente lo que facilita la concurrencia necesaria para mantener una interfaz de usuario receptiva.

Podemos utilizar este mismo principio para ejecutar dos operaciones asíncronas en paralelo:

```
var task1 = PrintAnswerToLife();
var task2 = PrintAnswerToLife();
await task1; await task2;
```

(Al esperar ambas operaciones posteriormente, "finalizamos" el paralelismo en ese punto. Más adelante, describiremos cómo el combinador de tareas WhenAll ayuda con este patrón).

La concurrencia creada de esta manera ocurre tanto si las operaciones se inician en un hilo de interfaz de usuario como si no, aunque hay una diferencia en cómo ocurre. En ambos casos, obtenemos la misma "verdadera" concurrencia que ocurre en las operaciones de nivel inferior que la inician (como Task.Delay o código enviado a Task.Run). Los métodos por encima de esto en la pila de llamadas estarán sujetos a una verdadera concurrencia solo si la operación se inició sin un contexto de sincronización presente; de lo contrario, estarán sujetos a la pseudo-concurrencia (y la simplicidad de la seguridad de hilos) de la que hablamos anteriormente, donde el único lugar en el que podemos ser interrumpidos es una declaración await. Esto nos permite, por ejemplo, definir un campo compartido, _x, e incrementarlo en GetAnswerToLife sin necesidad de bloquearlo:

```
async Task<int> GetAnswerToLife()
{
  _x++;
  await Task.Delay (5000);
  return 21 * 2;
}
```

(Sin embargo, no podríamos asumir que _x tuviera el mismo valor antes y después del await).

Expresiones lambda asíncronas

Al igual que los métodos con nombre pueden ser asíncronos:

```
async Task NamedMethod()
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
}
```

también pueden serlo los métodos sin nombre (expresiones lambda y métodos anónimos), si se les antepone la palabra clave async:

```
Func<Task> unnamed = async () =>
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
};
```

Podemos llamar y esperar estos métodos de la misma manera:

```
await NamedMethod();
await unnamed();
```

Podemos utilizar expresiones lambda asíncronas al adjuntar controladores de eventos:

```
myButton.Click += async (sender, args) =>
{
  await Task.Delay (1000);
  myButton.Content = "Done";
};
```

Esto es más conciso que el siguiente código, que tiene el mismo efecto:

```
myButton.Click += ButtonHandler;
...
async void ButtonHander (object sender, EventArgs args)
{
  await Task.Delay (1000);
  myButton.Content = "Done";
};
```

Las expresiones lambda asíncronas también pueden devolver Task<TResult>:

```
Func<Task<int>> unnamed = async () =>
{
  await Task.Delay (1000);
  return 123;
};
int answer = await unnamed();
```

Secuencias asíncronas

Con yield return, puedes escribir un iterador; con await, puedes escribir una función asíncrona. Las secuencias asíncronas (desde C# 8) combinan estos conceptos y te permiten escribir un iterador que espera, produciendo elementos de forma asíncrona. Este soporte se basa en el siguiente par de interfaces, que son los contrapartes asíncronos de las interfaces de enumeración que describimos en "Enumeración e Iteradores" en la página 189:

```
public interface IAsyncEnumerable<out T>
{
  IAsyncEnumerator<T> GetAsyncEnumerator (...);
}

public interface IAsyncEnumerator<out T>: IAsyncDisposable
{
  T Current { get; }
  ValueTask<bool> MoveNextAsync();
}
```

ValueTask<T> es una estructura que envuelve a Task<T> y tiene un comportamiento similar a Task<T>, al tiempo que permite una ejecución más eficiente cuando la tarea se completa de forma síncrona (lo cual ocurre con frecuencia al enumerar una secuencia). Consulta "ValueTask<T>" en la página 657 para obtener una discusión sobre las diferencias. IAsyncDisposable es una versión asíncrona de IDisposable; proporciona una oportunidad para realizar tareas de limpieza en caso de que elijas implementar las interfaces manualmente:

```
public interface IAsyncDisposable
{
  ValueTask DisposeAsync();
}
```



La acción de obtener cada elemento de la secuencia (MoveNextAsync) es una operación asíncrona, por lo que las secuencias asíncronas son adecuadas cuando los elementos llegan de forma gradual (como al procesar datos de un flujo de video). En contraste, el siguiente tipo es más adecuado cuando la secuencia en su conjunto se retrasa, pero los elementos, cuando llegan, llegan todos juntos:

```
Task<IEnumerable<T>>
```

Para generar una secuencia asíncrona, debes escribir un método que combine los principios de los iteradores y los métodos asíncronos. En otras palabras, tu método debe incluir tanto yield return como await, y debe devolver IAsyncEnumerable<T>:

```
async IAsyncEnumerable<int> RangeAsync (
  int start, int count, int delay)
{
  for (int i = start; i < start + count; i++)
  {
    await Task.Delay (delay);
    yield return i;
  }
}
```

Para consumir una secuencia asíncrona, utiliza la instrucción "await foreach":

```
await foreach (var number in RangeAsync (0, 10, 500))
  Console.WriteLine (number);
```

Ten en cuenta que los datos llegan constantemente, cada 500 milisegundos (o en la vida real, a medida que están disponibles). Esto contrasta con una construcción similar que utiliza Task<IEnumerable<T>>, donde no se devuelven datos hasta que esté disponible el último dato:

```
static async Task<IEnumerable<int>> RangeTaskAsync (int start, int count,
                                                    int delay)
{
  List<int> data = new List<int>();
  for (int i = start; i < start + count; i++)
  {
    await Task.Delay (delay);
    data.Add (i);
  }

  return data;
}
```

Aquí está cómo consumirlo con la instrucción foreach:

```
foreach (var data in await RangeTaskAsync(0, 10, 500))
  Console.WriteLine (data);
```

Consulta de IAsyncEnumerable<T>

El paquete NuGet System.Linq.Async define operadores de consulta LINQ que operan sobre IAsyncEnumerable<T>, lo que te permite escribir consultas de manera similar a IEnumerable<T>.

Por ejemplo, podemos escribir una consulta LINQ sobre el método RangeAsync que definimos en la sección anterior, de la siguiente manera:

```
IAsyncEnumerable<int> query =
  from i in RangeAsync (0, 10, 500)
  where i % 2 == 0   // Even numbers only.
  select i * 10;     // Multiply by 10.

await foreach (var number in query)
  Console.WriteLine (number);
```

Esto mostrará 0, 20, 40, y así sucesivamente.



Si estás familiarizado con Rx, también puedes beneficiarte de sus operadores de consulta (más potentes) llamando al método de extensión ToObservable, que convierte un IAsyncEnumerable<T> en un IObservable<T>. También está disponible el método de extensión ToAsyncEnumerable, para convertir en la dirección inversa.

IAsyncEnumerable<T> en ASP.Net Core

Las acciones del controlador de ASP.Net Core ahora pueden devolver IAsyncEnumerable<T>. Dichos métodos deben estar marcados como async. Por ejemplo:

```
[HttpGet]
public async IAsyncEnumerable<string> Get()
{
```

```
    using var dbContext = new BookContext();
    await foreach (var title in dbContext.Books
                                        .Select(b => b.Title)
                                        .AsAsyncEnumerable())
        yield return title;
}
```

Métodos asíncronos en WinRT

Si estás desarrollando aplicaciones UWP, deberás trabajar con los tipos de WinRT definidos en el sistema operativo. El equivalente de Task en WinRT es IAsyncAction, y el equivalente de Task<TResult> es IAsyncOperation<TResult>. Y para las operaciones que informan sobre el progreso, los equivalentes son IAsyncActionWithProgress<TProgress> e IAsyncOperationWithProgress<TResult, TProgress>. Todos ellos están definidos en el espacio de nombres Windows.Foundation.

Puedes convertir cualquiera de ellos en un Task o Task<TResult> mediante el método de extensión AsTask:

```
Task<StorageFile> fileTask = KnownFolders.DocumentsLibrary.CreateFileAsync
                             ("test.txt").AsTask();
```

O puedes esperarlos directamente:

```
StorageFile file = await KnownFolders.DocumentsLibrary.CreateFileAsync
                           ("test.txt");
```



Debido a limitaciones en el sistema de tipos COM, IAsyncActionWithProgress<TProgress> e IAsyncOperationWithProgress<TResult, TProgress> no se basan en IAsyncAction como podrías esperar. En cambio, ambos heredan de un tipo base común llamado IAsyncInfo.

El método AsTask también está sobrecargado para aceptar un token de cancelación (ver "Cancelación" en la página 659). También puede aceptar un objeto IProgress<T> cuando se encadena a las variantes WithProgress (ver "Informes de progreso" en la página 661).

Asincronía y contextos de sincronización

Ya hemos visto cómo la presencia de un contexto de sincronización es significativa en cuanto a la publicación de continuaciones. Hay un par de formas más sutiles en las que dichos contextos de sincronización entran en juego con funciones asincrónicas que devuelven void. Estas no son un resultado directo de las expansiones del compilador de C#, sino que son una función de los tipos Async*MethodBuilder en el espacio de nombres System.CompilerServices que el compilador utiliza al expandir las funciones asincrónicas.

Publicación de excepciones (Exception posting)

La publicación de excepciones en funciones asincrónicas que devuelven void presenta un desafío adicional. En aplicaciones de cliente rico, es común confiar en el evento central de manejo de

excepciones (Application.DispatcherUnhandledException en WPF) para procesar excepciones no controladas lanzadas en el subproceso de la interfaz de usuario. En las aplicaciones de ASP.NET Core, un atributo personalizado ExceptionFilterAttribute en el método ConfigureServices de Startup.cs realiza una tarea similar. Internamente, funcionan invocando eventos de la interfaz de usuario (o en ASP.NET Core, la canalización de métodos de procesamiento de páginas) en su propio bloque try/catch.

Las funciones asincrónicas de nivel superior complican este escenario. Considera el siguiente controlador de eventos para un clic de botón:

```
async void ButtonClick (object sender, RoutedEventArgs args)
{
  await Task.Delay(1000);
  throw new Exception ("Will this be ignored?");
}
```

Cuando se hace clic en el botón y se ejecuta el controlador de eventos, la ejecución vuelve normalmente al bucle de mensajes después de la instrucción await, y la excepción que se lanza un segundo después no puede ser capturada por el bloque catch en el bucle de mensajes.

Para mitigar este problema, AsyncVoidMethodBuilder captura las excepciones no controladas (en funciones asincrónicas que devuelven void) y las publica en el contexto de sincronización, si está presente, asegurando que los eventos globales de manejo de excepciones se ejecuten.



El compilador aplica esta lógica solo a las funciones asincrónicas que devuelven void. Si cambiamos ButtonClick para que devuelva una Task en lugar de void, la excepción no controlada provocará el fallo de la Task resultante, que luego no tendrá a dónde ir (lo que resultará en una excepción no observada).

Un matiz interesante es que no importa si se lanza la excepción antes o después de una instrucción await. Por lo tanto, en el siguiente ejemplo, la excepción se publica en el contexto de sincronización (si está presente) y nunca se devuelve al llamador:

```
async void Foo() { throw null; await Task.Delay(1000); }
```

(Si no hay un contexto de sincronización presente, la excepción se propagará en el grupo de subprocesos, lo que terminará la aplicación).

La razón por la que la excepción no se lanza directamente de vuelta al llamador es garantizar la previsibilidad y consistencia. En el siguiente ejemplo, la InvalidOperationException siempre tendrá el mismo efecto de provocar el fallo de la Task resultante, independientemente de someCondition:

```
async Task Foo()
{
  if (someCondition) await Task.Delay (100);
  throw new InvalidOperationException();
}
```

Los iteradores funcionan de manera similar:

```
IEnumerable<int> Foo() { throw null; yield return 123; }
```

En este ejemplo, la excepción no se lanza directamente de vuelta al llamador: no es hasta que se enumera la secuencia que se lanza la excepción.

OperationStarted y OperationCompleted

Si hay un contexto de sincronización presente, las funciones asíncronas que no devuelven ningún valor también llaman a su método OperationStarted al entrar en la función, y a su método OperationCompleted cuando la función termina.

Sobrescribir estos métodos es útil si se está escribiendo un contexto de sincronización personalizado para probar métodos asíncronos que no devuelven ningún valor. Esto se discute en el blog de Programación Paralela de Microsoft.

Optimizaciones

Finalización síncrona

Una función asíncrona puede devolver antes de esperar. Considera el siguiente método que almacena en caché la descarga de páginas web:

```
static Dictionary<string,string> _cache = new Dictionary<string,string>();

async Task<string> GetWebPageAsync (string uri)
{
  string html;
  if (_cache.TryGetValue (uri, out html)) return html;
  return _cache [uri] =
    await new WebClient().DownloadStringTaskAsync (uri);
}
```

Si un URI ya existe en la caché, la ejecución vuelve al llamador sin que se haya esperado, y el método devuelve una tarea que ya se ha completado. Esto se conoce como finalización síncrona.

Cuando esperas una tarea que se ha completado de forma síncrona, la ejecución no vuelve al llamador y rebota a través de una continuación; en cambio, pasa inmediatamente a la siguiente instrucción. El compilador implementa esta optimización comprobando la propiedad IsCompleted en el awaiter; en otras palabras, cada vez que esperas

```
Console.WriteLine (await GetWebPageAsync ("http://oreilly.com"));
```

el compilador emite código para omitir la continuación en caso de finalización de sincronización:

```
var awaiter = GetWebPageAsync().GetAwaiter();
if (awaiter.IsCompleted)
  Console.WriteLine (awaiter.GetResult());
else
  awaiter.OnCompleted (() => Console.WriteLine (awaiter.GetResult());
```



Esperar una función asíncrona que devuelve de manera síncrona todavía incurre en una sobrecarga (muy) pequeña, tal vez de 20 nanosegundos en una PC de la era 2019.

En contraste, el cambio al grupo de hilos introduce el costo de un cambio de contexto, tal vez de uno o dos microsegundos, y el cambio a un bucle de mensajes de la interfaz de usuario, al menos 10 veces eso (mucho más tiempo si el hilo de la interfaz de usuario está ocupado).

Incluso es legal escribir métodos asíncronos que nunca esperen, aunque el compilador generará una advertencia:

```
async Task<string> Foo() { return "abc"; }
```

Estos métodos pueden ser útiles al anular métodos virtuales/abstractos, si tu implementación no requiere asincronía. (Un ejemplo son los métodos ReadAsync/WriteAsync de MemoryStream; consulta el Capítulo 15). Otra forma de lograr el mismo resultado es usar Task.FromResult, que devuelve una tarea ya señalada:

```
Task<string> Foo() { return Task.FromResult ("abc"); }
```

Nuestro método GetWebPageAsync es implícitamente seguro para subprocesos si se llama desde un hilo de interfaz de usuario, en el sentido de que podrías invocarlo varias veces en sucesión (iniciando así múltiples descargas concurrentes) y no se requiere bloqueo para proteger la caché. Sin embargo, si las llamadas se realizaran al mismo URI, terminaríamos iniciando múltiples descargas redundantes, todas las cuales eventualmente actualizarían la misma entrada de la caché (siendo la última la que gane). Aunque no sea incorrecto, sería más eficiente si las llamadas subsiguientes al mismo URI pudieran esperar (de manera asíncrona) el resultado de la solicitud en curso.

Existe una manera fácil de lograr esto, sin recurrir a bloqueos o estructuras de señalización. En lugar de una caché de cadenas, creamos una caché de "futuros" (Task<string>):

```
static Dictionary<string,Task<string>> _cache =
   new Dictionary<string,Task<string>>();

Task<string> GetWebPageAsync (string uri)
{
  if (_cache.TryGetValue (uri, out var downloadTask)) return downloadTask;
  return _cache [uri] = new WebClient().DownloadStringTaskAsync (uri);
}
```

(Observa que no marcamos el método como async, porque estamos devolviendo directamente la tarea que obtenemos al llamar al método de WebClient).

Si llamamos a GetWebPageAsync repetidamente con el mismo URI, ahora tenemos la garantía de obtener el mismo objeto Task<string>. (Esto tiene el beneficio adicional de minimizar la carga de recolección de basura). Y si la tarea está completa, esperarla es barato, gracias a la optimización del compilador que acabamos de discutir.

Podríamos extender aún más nuestro ejemplo para hacerlo seguro para subprocesos sin la protección de un contexto de sincronización, bloqueando todo el cuerpo del método:

```
lock (_cache)
  if (_cache.TryGetValue (uri, out var downloadTask))
    return downloadTask;
  else
    return _cache [uri] = new WebClient().DownloadStringTaskAsync (uri);
}
```

Esto funciona porque no estamos bloqueando durante la descarga de una página (lo cual afectaría a la concurrencia); estamos bloqueando durante la breve duración de comprobar la caché, iniciar una nueva tarea si es necesario y actualizar la caché con esa tarea.

ValueTask<T>



ValueTask<T> está destinado a escenarios de microoptimización, y es posible que nunca necesites escribir métodos que devuelvan este tipo. Sin embargo, aún es importante tener en cuenta las precauciones que describiremos en la siguiente sección, porque algunos métodos de .NET devuelven ValueTask<T> y también se utiliza en IAsyncEnumerable<T>.

Acabamos de describir cómo el compilador optimiza una expresión await en una tarea completada de manera síncrona, al evitar la continuación y proceder inmediatamente a la siguiente instrucción. Si la finalización síncrona se debe al almacenamiento en caché, vimos que almacenar en caché la tarea en sí puede proporcionar una solución elegante y eficiente.

Sin embargo, no es práctico almacenar en caché la tarea en todos los escenarios de finalización síncrona. A veces, se debe crear una nueva tarea y esto crea una (pequeña) posible ineficiencia. Esto se debe a que Task y Task<T> son tipos de referencia, por lo que la creación requiere una asignación de memoria en el montón y una posterior recolección.

Una forma extrema de optimización es escribir código sin asignaciones; es decir, que no instancia ningún tipo de referencia y no agrega ninguna carga a la recolección de basura. Para admitir este patrón, se han introducido las estructuras ValueTask y ValueTask<T>, que el compilador permite utilizar en lugar de Task y Task<T>:

```
async ValueTask<int> Foo() { ... }
```

Esperar a un ValueTask<T> no implica asignaciones si la operación se completa de manera síncrona:

```
int answer = await Foo();   // (Potentially) allocation-free
```

Si la operación no se completa de manera síncrona, ValueTask<T> crea una Task<T> ordinaria en segundo plano (a la cual se le reenvía el await) y no se obtiene ningún beneficio. Puedes convertir un ValueTask<T> en una Task<T> ordinaria llamando al método AsTask.

También hay una versión no genérica, ValueTask, que es similar a Task.

Precauciones al usar ValueTask<T>

ValueTask<T> es relativamente inusual en el sentido de que se define como una estructura (struct) puramente por razones de rendimiento. Esto significa que está limitado por semánticas inapropiadas de tipo de valor que pueden llevar a sorpresas. Para evitar un comportamiento incorrecto, debes evitar lo siguiente:

- Hacer await en la misma ValueTask<T> varias veces.
- Llamar a .GetAwaiter().GetResult() cuando la operación no ha finalizado.

Si necesitas realizar estas acciones, llama a .AsTask() y opera en la Task resultante en su lugar.



La forma más sencilla de evitar estos problemas es hacer await directamente en una llamada al método, por ejemplo:

```
await Foo();   // Safe
```

La puerta al comportamiento erróneo se abre cuando asignas la tarea (value task) a una variable:

```
ValueTask<int> valueTask = Foo();  // Caution!
// Our use of valueTask can now lead to errors.
```

Esto se puede mitigar convirtiéndolo inmediatamente en una tarea ordinaria:

```
Task<int> task = Foo().AsTask();   // Safe
// task is safe to work with.
```

Evitar el rebote excesivo

Para métodos que se llaman muchas veces en un bucle, puedes evitar el costo de rebotar repetidamente en un bucle de mensajes de la interfaz de usuario (UI) llamando a ConfigureAwait. Esto fuerza a la tarea a no rebotar continuaciones en el contexto de sincronización, reduciendo la sobrecarga al costo de un cambio de contexto (o mucho menos si el método que estás esperando se completa de manera síncrona):

```
async void A() { ... await B(); ... }

async Task B()
{
  for (int i = 0; i < 1000; i++)
    await C().ConfigureAwait (false);
}

async Task C() { ... }
```

Esto significa que para los métodos B y C, abandonamos el modelo simple de seguridad de subproceso en aplicaciones de interfaz de usuario (UI) donde el código se ejecuta en el hilo de

la UI y solo puede ser interrumpido durante una instrucción await. Sin embargo, el método A no se ve afectado y permanecerá en un hilo de UI si se inició en uno.

Esta optimización es especialmente relevante al escribir bibliotecas, ya que no necesitas beneficiarte de una seguridad de subproceso simplificada porque tu código normalmente no comparte estado con el llamador y no accede a controles de la UI. (También tendría sentido, en nuestro ejemplo, que el método C se completara de manera síncrona si supiera que es probable que la operación sea de corta duración).

Patrones Asincrónicos (Asynchronous Patterns)

Cancelacion (Cancellation)

Es importante poder cancelar una operación concurrente después de que se haya iniciado, tal vez en respuesta a una solicitud del usuario. Una forma sencilla de implementar esto es con una bandera de cancelación, que podríamos encapsular escribiendo una clase como esta:

```
class CancellationToken
{
  public bool IsCancellationRequested { get; private set; }
  public void Cancel() { IsCancellationRequested = true; }
  public void ThrowIfCancellationRequested()
  {
    if (IsCancellationRequested)
      throw new OperationCanceledException();
  }
}
```

Luego, podríamos escribir un método asíncrono cancelable de la siguiente manera:

```
async Task Foo (CancellationToken cancellationToken)
{
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine (i);
    await Task.Delay (1000);
    cancellationToken.ThrowIfCancellationRequested();
  }
}
```

Cuando el llamador desea cancelar, llama a Cancel en el token de cancelación que pasó a Foo. Esto establece IsCancellationRequested en true, lo que hace que Foo falle poco tiempo después con una OperationCanceledException (una excepción predefinida en el espacio de nombres System diseñada para este propósito).

A parte de la seguridad de subprocesos (deberíamos bloquear alrededor de la lectura/escritura de IsCancellationRequested), este patrón es efectivo y el CLR proporciona un tipo llamado CancellationToken que es muy similar a lo que acabamos de mostrar. Sin embargo, carece de un método Cancel; este método se expone en otro tipo llamado CancellationTokenSource. Esta separación proporciona cierta seguridad: un método que solo tiene acceso a un objeto CancellationToken puede verificar pero no iniciar la cancelación.

Para obtener un token de cancelación, primero instanciamos un CancellationTokenSource:

```
var cancelSource = new CancellationTokenSource();
```

Esto expone una propiedad Token, que devuelve un CancellationToken. Por lo tanto, podríamos llamar a nuestro método Foo de la siguiente manera:

```
var cancelSource = new CancellationTokenSource();
Task foo = Foo (cancelSource.Token);
...

... (sometime later)
cancelSource.Cancel();
```

La mayoría de los métodos asíncronos en el CLR admiten tokens de cancelación, incluido Delay. Si modificamos Foo para que pase su token al método Delay, la tarea finalizará inmediatamente cuando se solicite (en lugar de hasta un segundo después):

```
async Task Foo (CancellationToken cancellationToken)
{
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine (i);
    await Task.Delay (1000, cancellationToken);
  }
}
```

Observa que ya no es necesario llamar a ThrowIfCancellationRequested porque Task.Delay se encarga de eso por nosotros. Los tokens de cancelación se propagan correctamente a lo largo de la pila de llamadas (así como las solicitudes de cancelación se propagan hacia arriba en la pila de llamadas mediante excepciones).



UWP utiliza tipos de WinRT, cuyos métodos asíncronos siguen un protocolo inferior para la cancelación, en el cual en lugar de aceptar un CancellationToken, el tipo IAsyncInfo expone un método Cancel. Sin embargo, el método de extensión AsTask está sobrecargado para aceptar un token de cancelación, lo que sirve como puente.

Los métodos síncronos también pueden admitir cancelación (como el método Wait de Task). En tales casos, la instrucción para cancelar debe venir de forma asíncrona (por ejemplo, desde otra tarea). Por ejemplo:

```
var cancelSource = new CancellationTokenSource();
Task.Delay (5000).ContinueWith (ant => cancelSource.Cancel());
...
```

De hecho, al construir CancellationTokenSource, puedes especificar un intervalo de tiempo para iniciar la cancelación después de un período establecido (tal como lo demostramos). Esto es útil para implementar límites de tiempo (timeouts), ya sea en métodos síncronos o asíncronos:

```
var cancelSource = new CancellationTokenSource (5000);
try { await Foo (cancelSource.Token); }
catch (OperationCanceledException ex) { Console.WriteLine ("Cancelled"); }
```

La estructura CancellationToken proporciona un método Register que te permite registrar un delegado de devolución de llamada que se ejecutará al producirse la cancelación. Este método devuelve un objeto que se puede descartar para deshacer el registro.

Las tareas generadas por las funciones asíncronas del compilador entran automáticamente en un estado "Canceled" cuando se produce una excepción de cancelación no controlada (IsCanceled devuelve true y IsFaulted devuelve false). Lo mismo ocurre con las tareas creadas con Task.Run a las que se les pasa el mismo CancellationToken al constructor. La distinción entre una tarea fallida y una tarea cancelada no es importante en escenarios asíncronos, ya que ambas lanzan una excepción OperationCanceledException cuando se les espera; esto es importante en escenarios avanzados de programación paralela (específicamente, con continuaciones condicionales). Abordamos este tema en "Canceling Tasks" en la página 933.

Informar sobre el progreso

A veces, quieres que una operación asíncrona informe sobre el progreso a medida que se ejecuta. Una solución simple es pasar un delegado Action al método asíncrono, que el método invoca cada vez que cambia el progreso:

```
Task Foo (Action<int> onProgressPercentChanged)
{
  return Task.Run (() =>
  {
    for (int i = 0; i < 1000; i++)
    {
      if (i % 10 == 0) onProgressPercentChanged (i / 10);
      // Do something compute-bound...
    }
  });
}
```

Así es cómo podríamos llamarlo:

```
Action<int> progress = i => Console.WriteLine (i + " %");
await Foo (progress);
```

Aunque esto funciona bien en una aplicación de consola, no es ideal en escenarios de aplicaciones de cliente avanzadas porque informa sobre el progreso desde un subproceso de trabajo, lo que puede causar problemas de seguridad en el subproceso principal. En efecto, hemos permitido que un efecto secundario de la concurrencia "filtre" hacia el mundo exterior, lo cual es desafortunado dado que el método, por lo demás, está aislado si se llama desde un subproceso de interfaz de usuario (UI).

IProgress<T> y Progress<T>

El CLR proporciona un par de tipos para resolver este problema: una interfaz llamada IProgress<T> y una clase que implementa esta interfaz llamada Progress<T>. Su propósito, en esencia, es "envolver" un delegado para que las aplicaciones de interfaz de usuario puedan informar sobre el progreso de forma segura a través del contexto de sincronización.

La interfaz define solo un método:

```
public interface IProgress<in T>
{
  void Report (T value);
}
```

Usar IProgress<T> es fácil; nuestro método apenas cambia:

```
Task Foo (IProgress<int> onProgressPercentChanged)
{
  return Task.Run (() =>
  {
    for (int i = 0; i < 1000; i++)
    {
      if (i % 10 == 0) onProgressPercentChanged.Report (i / 10);


        // Do something compute-bound...
      }
    });
  }
```

La clase Progress<T> tiene un constructor que acepta un delegado de tipo Action<T> que envuelve:

```
var progress = new Progress<int> (i => Console.WriteLine (i + " %"));
await Foo (progress);
```

(Progress<T> también tiene un evento ProgressChanged al que puedes suscribirte en lugar de (o además de) pasar un delegado de acción al constructor). Al instanciar Progress<int>, la clase captura el contexto de sincronización, si está presente. Cuando Foo llama a Report, el delegado se invoca a través de ese contexto.

Los métodos asíncronos pueden implementar una información de progreso más elaborada reemplazando int con un tipo personalizado que exponga una variedad de propiedades.



Si estás familiarizado con Rx, notarás que IProgress<T> junto con la tarea devuelta por la función asíncrona proporcionan un conjunto de características similar a IObserver<T>. La diferencia es que una tarea puede exponer un valor de retorno "final" además de (y de tipo diferente a) los valores emitidos por IProgress<T>. Los valores emitidos por IProgress<T> son típicamente valores "desechables" (por ejemplo, porcentaje completo o bytes descargados hasta ahora), mientras que los valores enviados por OnNext de IObserver<T> generalmente comprenden el propio resultado y son la razón principal para llamarlo.

Los métodos asíncronos en WinRT también ofrecen informes de progreso, aunque el protocolo se complica por el (relativamente) sistema de tipos primitivo de COM. En lugar de aceptar un objeto IProgress<T>, los métodos asincrónicos de WinRT que informan sobre el progreso devuelven una de las siguientes interfaces, en lugar de IAsyncAction e IAsyncOperation<TResult>:

```
IAsyncActionWithProgress<TProgress>
IAsyncOperationWithProgress<TResult, TProgress>
```

Curiosamente, ambas se basan en IAsyncInfo (no en IAsyncAction e IAsyncOperation<TResult>).

La buena noticia es que el método de extensión AsTask también está sobrecargado para aceptar IProgress<T> para las interfaces mencionadas anteriormente, por lo que como consumidor de .NET, puedes ignorar las interfaces de COM y hacer esto:

```
var progress = new Progress<int> (i => Console.WriteLine (i + " %"));
CancellationToken cancelToken = ...
var task = someWinRTobject.FooAsync().AsTask (cancelToken, progress);
```

El Patrón Asincrónico Basado en Tareas (Task-Based Asynchronous Pattern)

.NET expone cientos de métodos asíncronos que devuelven tareas que puedes esperar (principalmente relacionados con E/S). La mayoría de estos métodos siguen (al menos en parte) un patrón llamado Patrón Asincrónico Basado en Tareas (TAP), que es una formalización sensata de lo que hemos descrito hasta ahora. Un método TAP hace lo siguiente:

- Devuelve una tarea "activa" (en ejecución) (Task o Task<TResult>).
- Tiene un sufijo "Async" (excepto en casos especiales como los combinadores de tareas).
- Está sobrecargado para aceptar un token de cancelación y/o un objeto IProgress<T> si admite cancelación y/o informe de progreso.
- Vuelve rápidamente al llamador (tiene solo una pequeña fase sincrónica inicial).
- No bloquea un hilo si está relacionado con E/S.

Como hemos visto, los métodos TAP son fáciles de escribir con las funciones asíncronas de C#.

Combinadores de tareas

Una consecuencia positiva de que exista un protocolo consistente para las funciones asíncronas (en el que devuelven tareas de manera consistente) es que es posible usar y escribir combinadores de tareas, es decir, funciones que combinan de manera útil tareas sin tener en cuenta lo que hacen específicamente esas tareas.

El CLR incluye dos combinadores de tareas: Task.WhenAny y Task.WhenAll. Al describirlos, supondremos que se definen los siguientes métodos:

```
async Task<int> Delay1() { await Task.Delay (1000); return 1; }
async Task<int> Delay2() { await Task.Delay (2000); return 2; }
async Task<int> Delay3() { await Task.Delay (3000); return 3; }
```

WhenAny

Task.WhenAny devuelve una tarea que se completa cuando cualquiera de un conjunto de tareas se completa. Lo siguiente se completa en un segundo:

```
Task<int> winningTask = await Task.WhenAny (Delay1(), Delay2(), Delay3());
Console.WriteLine ("Done");
Console.WriteLine (winningTask.Result);   // 1
```

Debido a que Task.WhenAny en sí devuelve una tarea, lo esperamos con await, lo cual devuelve la tarea que se haya terminado primero. Nuestro ejemplo no bloquea en absoluto, incluso en la última línea cuando accedemos a la propiedad Result (porque tareaGanadora ya se habrá terminado).

No obstante, por lo general es mejor esperar directamente la tarea ganadora:

```
Console.WriteLine (await winningTask);   // 1
```

porque cualquier excepción se relanzará sin envolverla en un AggregateException. De hecho, podemos realizar ambos await en un solo paso:

```
int answer = await await Task.WhenAny (Delay1(), Delay2(), Delay3());
```

Si una tarea que no es la ganadora presenta un fallo posteriormente, la excepción pasará desapercibida a menos que posteriormente esperes la tarea (await) o consultes su propiedad Exception.

WhenAny es útil para aplicar límites de tiempo o cancelación a operaciones que de otra manera no lo admiten:

```
Task<string> task = SomeAsyncFunc();
Task winner = await (Task.WhenAny (task, Task.Delay(5000)));
if (winner != task) throw new TimeoutException();
string result = await task;   // Unwrap result/re-throw
```

Observa que, en este caso, al llamar a WhenAny con tareas de tipos diferentes, el ganador se reporta como una simple Task (en lugar de una Task<string>).

WhenAll

Task.WhenAll devuelve una tarea que se completa cuando todas las tareas que le pasas se completan. Lo siguiente se completa después de tres segundos (y demuestra el patrón de bifurcación/join):

```
await Task.WhenAll (Delay1(), Delay2(), Delay3());
```

Podríamos obtener un resultado similar esperando task1, task2 y task3 de forma individual en lugar de utilizar WhenAll:

```
Task task1 = Delay1(), task2 = Delay2(), task3 = Delay3();
await task1; await task2; await task3;
```

La diferencia (además de ser menos eficiente al requerir tres esperas en lugar de una) es que si task1 falla, nunca llegaremos a esperar task2/task3 y cualquiera de sus excepciones pasará desapercibida.

En contraste, Task.WhenAll no se completa hasta que todas las tareas se hayan completado, incluso si hay un fallo. Y si hay múltiples fallos, sus excepciones se combinan en la AggregateException de la tarea (esto es cuando AggregateException resulta útil, si estás interesado en todas las excepciones). Sin embargo, al esperar la tarea combinada, solo se lanza la primera excepción, por lo que para ver todas las excepciones debes hacer lo siguiente:

```
Task task1 = Task.Run (() => { throw null; } );
Task task2 = Task.Run (() => { throw null; } );
Task all = Task.WhenAll (task1, task2);
try { await all; }
catch
{
  Console.WriteLine (all.Exception.InnerExceptions.Count);   // 2
}
```

Cuando se llama a WhenAll con tareas del tipo Task<TResult>, devuelve una Task<TResult[]>, que proporciona los resultados combinados de todas las tareas. Esto se reduce a un TResult[] cuando se espera:

```
Task<int> task1 = Task.Run (() => 1);
Task<int> task2 = Task.Run (() => 2);
int[] results = await Task.WhenAll (task1, task2);   // { 1, 2 }
```

Para dar un ejemplo práctico, lo siguiente descarga URI en paralelo y suma su longitud total:

```
async Task<int> GetTotalSize (string[] uris)
{
  IEnumerable<Task<byte[]>> downloadTasks = uris.Select (uri =>
    new WebClient().DownloadDataTaskAsync (uri));

  byte[][] contents = await Task.WhenAll (downloadTasks);
  return contents.Sum (c => c.Length);
}
```

Sin embargo, hay una ligera ineficiencia aquí, ya que estamos manteniendo innecesariamente los arreglos de bytes descargados hasta que todas las tareas estén completas. Sería más eficiente si colapsáramos los arreglos de bytes en sus longitudes inmediatamente después de descargarlos. Aquí es donde entra en juego una lambda asíncrona, ya que necesitamos proporcionar una expresión await en el operador de consulta Select de LINQ:

```
async Task<int> GetTotalSize (string[] uris)
{
  IEnumerable<Task<int>> downloadTasks = uris.Select (async uri =>
    (await new WebClient().DownloadDataTaskAsync (uri)).Length);

  int[] contentLengths = await Task.WhenAll (downloadTasks);
  return contentLengths.Sum();
}
```

Combinadores personalizados

Puede ser útil escribir tus propios combinadores de tareas. El "combinador" más simple acepta una sola tarea, como el siguiente ejemplo, que te permite esperar cualquier tarea con un tiempo de espera:

```
async static Task<TResult> WithTimeout<TResult> (this Task<TResult> task,
                                                 TimeSpan timeout)
{
  Task winner = await Task.WhenAny (task, Task.Delay (timeout))
                        .ConfigureAwait (false);
  if (winner != task) throw new TimeoutException();
  return await task.ConfigureAwait (false);   // Unwrap result/re-throw
}
```

Dado que este es claramente un "método de biblioteca" que no accede a un estado compartido externo, utilizamos ConfigureAwait(false) al esperar para evitar posibles cambios de contexto de sincronización de la interfaz de usuario. Podemos mejorar aún más la eficiencia cancelando el Task.Delay cuando la tarea se completa a tiempo (esto evita la pequeña sobrecarga de mantener un temporizador activo):

```
async static Task<TResult> WithTimeout<TResult> (this Task<TResult> task,
                                                 TimeSpan timeout)
{


  var cancelSource = new CancellationTokenSource();
  var delay = Task.Delay (timeout, cancelSource.Token);
  Task winner = await Task.WhenAny (task, delay).ConfigureAwait (false);
  if (winner == task)
    cancelSource.Cancel();
  else
    throw new TimeoutException();
  return await task.ConfigureAwait (false);   // Unwrap result/re-throw
}
```

El siguiente código te permite "abandonar" una tarea mediante un CancellationToken:

```
static Task<TResult> WithCancellation<TResult> (this Task<TResult> task,
                                                CancellationToken cancelToken)
{
  var tcs = new TaskCompletionSource<TResult>();
  var reg = cancelToken.Register (() => tcs.TrySetCanceled ());
  task.ContinueWith (ant =>
  {
    reg.Dispose();
    if (ant.IsCanceled)
      tcs.TrySetCanceled();
    else if (ant.IsFaulted)
      tcs.TrySetException (ant.Exception.InnerException);
    else
      tcs.TrySetResult (ant.Result);
  });
  return tcs.Task;
}
```

Los combinadores de tareas pueden ser complejos de escribir, a veces requiriendo el uso de constructos de señalización, que se explican en el Capítulo 21. En realidad, esto es algo positivo, ya que mantiene la complejidad relacionada con la concurrencia fuera de la lógica de negocio y la encapsula en métodos reutilizables que pueden ser probados de forma aislada.

El siguiente combinador funciona como WhenAll, excepto que si alguna de las tareas falla, la tarea resultante fallará inmediatamente:

```
async Task<TResult[]> WhenAllOrError<TResult>
  (params Task<TResult>[] tasks)
{
  var killJoy = new TaskCompletionSource<TResult[]>();
  foreach (var task in tasks)
    task.ContinueWith (ant =>
    {
      if (ant.IsCanceled)
        killJoy.TrySetCanceled();
      else if (ant.IsFaulted)
        killJoy.TrySetException (ant.Exception.InnerException);
    });
  return await await Task.WhenAny (killJoy.Task, Task.WhenAll (tasks))
                         .ConfigureAwait (false);
}
```

Comenzamos creando un TaskCompletionSource cuya única tarea es finalizar la ejecución si una tarea falla. Por lo tanto, nunca llamamos a su método SetResult, solo llamamos a sus métodos TrySetCanceled y TrySetException. En este caso, ContinueWith es más conveniente que GetAwaiter().OnCompleted porque no estamos accediendo a los resultados de las tareas y no queremos cambiar a un hilo de interfaz de usuario en ese punto.

Bloqueo asíncrono

En "Semáforos y bloqueos asíncronos" en la página 881, describimos cómo usar SemaphoreSlim para bloquear o limitar la concurrencia de forma asíncrona.

Patrones obsoletos

.NET utiliza otros patrones de asincronía que preceden a las tareas y a las funciones asíncronas. Estos rara vez se requieren ahora que la asincronía basada en tareas se ha convertido en el patrón dominante.

Modelo de programación asíncrona

El patrón más antiguo se llama Modelo de Programación Asíncrona (APM) y utiliza un par de métodos que comienzan con "Begin" y "End", y una interfaz llamada IAsyncResult. Para ilustrar, tomemos la clase Stream en System.IO y veamos su método Read. Primero, la versión síncrona:

```
public int Read (byte[] buffer, int offset, int size);
```

Probablemente puedas predecir cómo se ve la versión basada en tareas asíncronas:

```
public Task<int> ReadAsync (byte[] buffer, int offset, int size);
```

Ahora examinemos la versión APM:

```
public IAsyncResult BeginRead (byte[] buffer, int offset, int size,
                               AsyncCallback callback, object state);
public int EndRead (IAsyncResult asyncResult);
```

Llamar al método Begin* inicia la operación, devolviendo un objeto IAsyncResult que actúa como un token para la operación asíncrona. Cuando la operación se completa (o falla), se activa el delegado AsyncCallback:

```
public delegate void AsyncCallback (IAsyncResult ar);
```

Quien maneje este delegado luego llama al método End*, que proporciona el valor de retorno de la operación y relanza una excepción si la operación falló.

El APM no solo es incómodo de usar, sino que también es sorprendentemente difícil de implementar correctamente. La forma más sencilla de lidiar con los métodos APM es llamar al método adaptador Task.Factory.FromAsync, que convierte un par de métodos APM en un Task. Internamente, utiliza un TaskCompletionSource para proporcionarte una tarea que se activa cuando una operación APM se completa o falla.

El método FromAsync requiere los siguientes parámetros:

- Un delegado que especifica un método BeginXXX.
- Un delegado que especifica un método EndXXX.
- Argumentos adicionales que se pasarán a estos métodos.

FromAsync está sobrecargado para aceptar tipos de delegados y argumentos que coinciden con casi todas las firmas de métodos asíncronos que se encuentran en .NET. Por ejemplo, suponiendo que "stream" es un Stream y "buffer" es un byte[], podríamos hacer lo siguiente:

```
Task<int> readChunk = Task<int>.Factory.FromAsync (
  stream.BeginRead, stream.EndRead, buffer, 0, 1000, null);
```

Patrón Asincrónico Basado en Eventos

El Patrón Asincrónico Basado en Eventos (EAP, por sus siglas en inglés) se introdujo en 2005 para proporcionar una alternativa más sencilla al APM, especialmente en escenarios de interfaz de usuario. Sin embargo, solo se implementó en unos pocos tipos, especialmente en WebClient en System.Net. El EAP es solo un patrón; no se proporcionan tipos específicos para ayudar. Básicamente, el patrón es el siguiente: una clase ofrece una serie de miembros que gestionan internamente la concurrencia, similar a lo siguiente:

```
// These members are from the WebClient class:

public byte[] DownloadData (Uri address);    // Synchronous version
public void DownloadDataAsync (Uri address);
public void DownloadDataAsync (Uri address, object userToken);
public event DownloadDataCompletedEventHandler DownloadDataCompleted;

public void CancelAsync (object userState);  // Cancels an operation
public bool IsBusy { get; }                  // Indicates if still running
```

Los métodos *Async inician una operación de forma asíncrona. Cuando la operación se completa, se dispara el evento *Completed (enviándolo automáticamente al contexto de sincronización capturado si está presente). Este evento devuelve un objeto de argumentos de evento que contiene lo siguiente:

- Un indicador que indica si la operación fue cancelada (por el consumidor llamando a CancelAsync).
- Un objeto Error que indica la excepción que se produjo (si hubo alguna).
- El objeto userToken si se proporcionó al llamar al método Async.

Los tipos EAP también pueden exponer un evento de informe de progreso, que se dispara cada vez que cambia el progreso (también se publica a través del contexto de sincronización):

```
public event DownloadProgressChangedEventHandler DownloadProgressChanged;
```

Implementar el EAP requiere una gran cantidad de código redundante, lo que hace que el patrón sea poco componible.

BackgroundWorker

BackgroundWorker en System.ComponentModel es una implementación de propósito general del EAP. Permite que las aplicaciones de cliente rico inicien un hilo de trabajo y reporten la finalización y el progreso basado en porcentaje sin necesidad de capturar explícitamente el contexto de sincronización. Aquí tienes un ejemplo:

```
var worker = new BackgroundWorker { WorkerSupportsCancellation = true };
worker.DoWork += (sender, args) =>
{                                       // This runs on a worker thread
  if (args.Cancel) return;
  Thread.Sleep(1000);
  args.Result = 123;
};
worker.RunWorkerCompleted += (sender, args) =>
{                                                   // Runs on UI thread
  // We can safely update UI controls here...
  if (args.Cancelled)
    Console.WriteLine ("Cancelled");
  else if (args.Error != null)
    Console.WriteLine ("Error: " + args.Error.Message);
  else
    Console.WriteLine ("Result is: " + args.Result);
};
worker.RunWorkerAsync();   // Captures sync context and starts operation
```

El método RunWorkerAsync inicia la operación, disparando el evento DoWork en un hilo de trabajo de grupo. También captura el contexto de sincronización y, cuando la operación se completa (o falla), el evento RunWorkerCompleted se invoca a través de ese contexto de sincronización (como una continuación).

BackgroundWorker crea una concurrencia de granularidad gruesa, en el sentido de que el evento DoWork se ejecuta completamente en un hilo de trabajo. Si necesitas actualizar los controles de la interfaz de usuario en ese controlador de eventos (además de enviar un mensaje de porcentaje completado), debes utilizar Dispatcher.BeginInvoke u operaciones similares.

Proporcionamos más detalles sobre BackgroundWorker en http://albahari.com/threading.

15 Flujos y E/S (Streams and I/O)

Este capítulo describe los tipos fundamentales para la entrada y salida en .NET, con énfasis en los siguientes temas:

- La arquitectura de flujos en .NET y cómo proporciona una interfaz de programación consistente para la lectura y escritura en diversos tipos de E/S.
- Clases para manipular archivos y directorios en el disco.
- Flujos especializados para compresión, tuberías con nombre y archivos mapeados en memoria.

Este capítulo se centra en los tipos de espacio de nombres System.IO, que son el hogar de la funcionalidad de E/S de nivel inferior.

Arquitectura de flujos (Stream Architecture)

La arquitectura de flujos en .NET se centra en tres conceptos: almacenamiento subyacente, decoradores y adaptadores, como se muestra en la Figura 15-1.

Un almacenamiento subyacente es el punto final que hace que la entrada y salida sean útiles, como un archivo o una conexión de red. Precisamente, puede ser uno o ambos de los siguientes:

- Una fuente desde la cual se pueden leer secuencialmente bytes.
- Un destino al cual se pueden escribir secuencialmente bytes.

Sin embargo, un almacenamiento subyacente no tiene utilidad a menos que se exponga al programador. Un Stream es la clase estándar de .NET con este propósito; expone un conjunto estándar de métodos para leer, escribir y posicionar. A diferencia de una matriz, en la cual todos los datos subyacentes existen en la memoria de una vez, un flujo maneja los datos de manera serial, ya sea un byte a la vez o en bloques de un tamaño manejable. Por lo tanto, un flujo puede utilizar una cantidad pequeña y fija de memoria independientemente del tamaño de su almacenamiento subyacente.

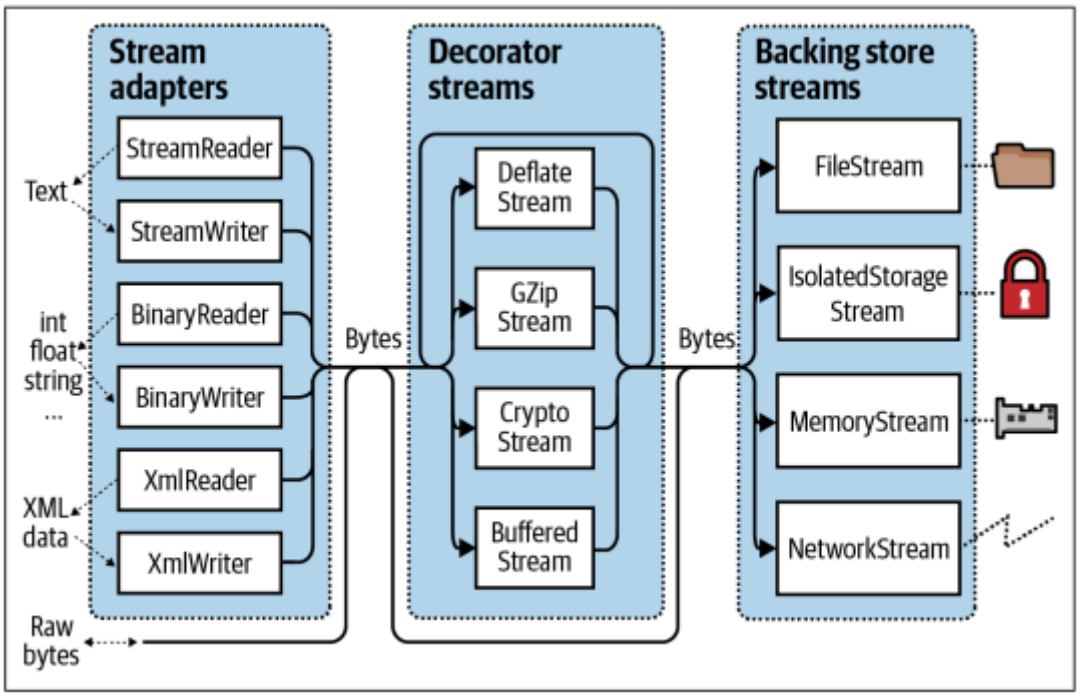


Figure 15-1. Stream architecture

Los flujos se dividen en dos categorías:

Flujos de almacenamiento subyacente

> Estos están diseñados para un tipo específico de almacenamiento subyacente, como FileStream o NetworkStream.

Flujos decoradores

> Estos se alimentan de otro flujo, transformando los datos de alguna manera, como DeflateStream o CryptoStream.

Los flujos decoradores tienen los siguientes beneficios arquitectónicos:

- Liberan a los flujos de almacenamiento subyacente de la necesidad de implementar características como compresión y cifrado.
- Los flujos no sufren cambios en su interfaz cuando se decoran.
- Puedes conectar decoradores en tiempo de ejecución.
- Puedes encadenar decoradores entre sí (por ejemplo, un compresor seguido de un cifrador).

Tanto los flujos de almacenamiento subyacente como los flujos decoradores trabajan exclusivamente con bytes. Aunque esto es flexible y eficiente, las aplicaciones a menudo trabajan a niveles superiores, como texto o XML. Los adaptadores cubren esta brecha envolviendo un flujo en una clase con métodos especializados de un formato particular. Por ejemplo, un lector de texto expone un método ReadLine; un escritor de XML expone un método WriteAttributes.



Un adaptador envuelve un flujo, al igual que un decorador. Sin embargo, a diferencia de un decorador, un adaptador no es en sí mismo un flujo; típicamente oculta completamente los métodos orientados a bytes.

En resumen, los flujos de almacenamiento subyacente proporcionan los datos en bruto, los flujos decoradores proporcionan transformaciones binarias transparentes como el cifrado, y los adaptadores ofrecen métodos tipados para trabajar con tipos de alto nivel como cadenas de texto y XML. La Figura 15-1 ilustra sus asociaciones. Para componer una cadena, simplemente pasas un objeto a través del constructor de otro.

Uso de los flujos (Using Streams)

La clase abstracta Stream es la base para todos los flujos. Define métodos y propiedades para tres operaciones fundamentales: lectura, escritura y búsqueda, así como tareas administrativas como cierre, vaciado y configuración de tiempos de espera (consultar Tabla 15-1).

Table 15-1. Stream class members

Category	Members
Reading	`public abstract bool CanRead { get; }`
	`public abstract int Read (byte[] buffer, int offset, int count)`
	`public virtual int ReadByte();`
Writing	`public abstract bool CanWrite { get; }`
	`public abstract void Write (byte[] buffer, int offset, int count);`
	`public virtual void WriteByte (byte value);`
Seeking	`public abstract bool CanSeek { get; }`
	`public abstract long Position { get; set; }`
	`public abstract void SetLength (long value);`
	`public abstract long Length { get; }`
	`public abstract long Seek (long offset, SeekOrigin origin);`
Closing/ flushing	`public virtual void Close();`
	`public void Dispose();`
	`public abstract void Flush();`
Timeouts	`public virtual bool CanTimeout { get; }`
	`public virtual int ReadTimeout { get; set; }`
	`public virtual int WriteTimeout { get; set; }`
Other	`public static readonly Stream Null; // "Null" stream`
	`public static Stream Synchronized (Stream stream);`

También existen versiones asincrónicas de los métodos Read y Write, que devuelven Tasks y opcionalmente aceptan un token de cancelación, así como sobrecargas que funcionan con los tipos Span<T> y Memory<T> que se describen en el Capítulo 23.

En el siguiente ejemplo, utilizamos un flujo de archivo para leer, escribir y buscar:

```
using System;
using System.IO;

// Create a file called test.txt in the current directory:
using (Stream s = new FileStream ("test.txt", FileMode.Create))
{
  Console.WriteLine (s.CanRead);       // True
  Console.WriteLine (s.CanWrite);      // True
  Console.WriteLine (s.CanSeek);       // True

  s.WriteByte (101);
  s.WriteByte (102);
  byte[] block = { 1, 2, 3, 4, 5 };
  s.Write (block, 0, block.Length);     // Write block of 5 bytes

  Console.WriteLine (s.Length);         // 7
  Console.WriteLine (s.Position);       // 7
  s.Position = 0;                       // Move back to the start

  Console.WriteLine (s.ReadByte());     // 101
  Console.WriteLine (s.ReadByte());     // 102

  // Read from the stream back into the block array:
  Console.WriteLine (s.Read (block, 0, block.Length));   // 5

  // Assuming the last Read returned 5, we'll be at
  // the end of the file, so Read will now return 0:
  Console.WriteLine (s.Read (block, 0, block.Length));   // 0
}
```

La lectura o escritura de forma asíncrona es simplemente cuestión de llamar a los métodos ReadAsync/WriteAsync en lugar de Read/Write, y esperar la expresión (también debemos agregar la palabra clave `async` al método de llamada, como se describió en el Capítulo 14):

```
async static void AsyncDemo()
{
  using (Stream s = new FileStream ("test.txt", FileMode.Create))
  {
    byte[] block = { 1, 2, 3, 4, 5 };
    await s.WriteAsync (block, 0, block.Length);    // Write asychronously

    s.Position = 0;                       // Move back to the start

    // Read from the stream back into the block array:
    Console.WriteLine (await s.ReadAsync (block, 0, block.Length));   // 5
  }
}
```

Los métodos asíncronos facilitan la escritura de aplicaciones receptivas y escalables que trabajan con flujos potencialmente lentos (especialmente flujos de red), sin bloquear un hilo.

 Por motivos de brevedad, seguiremos utilizando métodos síncronos para la mayoría de los ejemplos en este capítulo; sin embargo, recomendamos las operaciones asincrónicas de lectura/escritura como preferibles en la mayoría de los escenarios que involucran E/S de red.

Lectura y Escritura

Un flujo puede admitir la lectura, la escritura o ambas. Si CanWrite devuelve false, el flujo es de solo lectura; si CanRead devuelve false, el flujo es de solo escritura.

El método Read recibe un bloque de datos desde el flujo y lo almacena en una matriz. Devuelve el número de bytes recibidos, que siempre es menor o igual al argumento count. Si es menor que count, significa que se ha alcanzado el final del flujo o que el flujo está proporcionando los datos en fragmentos más pequeños (como suele ser el caso de los flujos de red). En ambos casos, los bytes restantes en la matriz permanecerán sin escribir, conservando sus valores anteriores.

 Con Read, solo puedes estar seguro de que has alcanzado el final del flujo cuando el método devuelve 0. Por lo tanto, si tienes un flujo de 1.000 bytes, el siguiente código podría no leerlo por completo en la memoria:

```
// Assuming s is a stream:
byte[] data = new byte [1000];
s.Read (data, 0, data.Length);
```

El método Read podría leer desde 1 hasta 1.000 bytes, dejando el resto del flujo sin leer.

Aquí tienes la forma correcta de leer un flujo de 1.000 bytes:

```
byte[] data = new byte [1000];

// bytesRead will always end up at 1000, unless the stream is
// itself smaller in length:

int bytesRead = 0;
int chunkSize = 1;
while (bytesRead < data.Length && chunkSize > 0)
  bytesRead +=
    chunkSize = s.Read (data, bytesRead, data.Length - bytesRead);
```

 Afortunadamente, el tipo BinaryReader proporciona una forma más sencilla de lograr el mismo resultado:

```
byte[] data = new BinaryReader (s).ReadBytes (1000);
```

Si el flujo es menor que 1.000 bytes de longitud, la matriz de bytes devuelta reflejará el tamaño real del flujo. Si el flujo es susceptible de búsqueda, puedes leer todo su contenido reemplazando 1000 por `(int)s.Length`.

Describimos el tipo BinaryReader con más detalle en "Adaptadores de flujo" en la página 687.

El método ReadByte es más simple: lee solo un byte y devuelve -1 para indicar el final del flujo. ReadByte en realidad devuelve un entero en lugar de un byte porque este último no puede devolver -1.

Los métodos Write y WriteByte envían datos al flujo. Si no pueden enviar los bytes especificados, se lanza una excepción.

 En los métodos Read y Write, el argumento offset se refiere al índice en la matriz de búfer donde comienza la lectura o escritura, no a la posición dentro del flujo.

Búsqueda (Seeking)

Un flujo es susceptible de búsqueda si CanSeek devuelve true. Con un flujo susceptible de búsqueda (como un flujo de archivo), puedes consultar o modificar su longitud (llamando a SetLength) y en cualquier momento cambiar la posición en la que estás leyendo o escribiendo. La propiedad Position es relativa al inicio del flujo; sin embargo, el método Seek te permite moverte relativo a la posición actual o al final del flujo.

 Cambiar la posición en un FileStream normalmente lleva unos pocos microsegundos. Si estás haciendo esto millones de veces en un bucle, la clase MemoryMappedFile podría ser una mejor opción que un FileStream (ver "Archivos mapeados en memoria" en la página 712).

Con un flujo no susceptible de búsqueda (como un flujo de cifrado), la única forma de determinar su longitud es leerlo por completo. Además, si necesitas volver a leer una sección anterior, debes cerrar el flujo y comenzar de nuevo con uno nuevo.

Cierre y Flushing (Closing and Flushing)

Los flujos deben ser liberados después de su uso para liberar los recursos subyacentes, como los identificadores de archivos y sockets. Una forma sencilla de garantizar esto es mediante la instanciación de flujos dentro de bloques using. En general, los flujos siguen las convenciones estándar de liberación:

- Dispose y Close son idénticos en función.
- Disponer o cerrar un flujo repetidamente no causa ningún error.

Cerrar un flujo decorador también cierra tanto el decorador como su flujo de almacenamiento subyacente. Con una cadena de decoradores, cerrar el decorador más externo (al inicio de la cadena) cierra todo el conjunto.

Algunos flujos almacenan en búfer internamente los datos hacia y desde el almacenamiento subyacente para reducir el tiempo de ida y vuelta y mejorar el rendimiento (los flujos de archivos son un buen ejemplo de esto). Esto significa que los datos que escribas en un flujo pueden no llegar inmediatamente al almacenamiento subyacente; pueden retrasarse a medida que el búfer se llena. El método Flush obliga a que cualquier dato almacenado en búfer se escriba de inmediato. Flush se llama automáticamente cuando se cierra un flujo, por lo que nunca necesitas hacer lo siguiente:

```
s.Flush(); s.Close();
```

Tiempos de espera (Timeouts)

Un flujo admite tiempos de espera de lectura y escritura si CanTimeout devuelve true. Los flujos de red admiten tiempos de espera, mientras que los flujos de archivos y de memoria no lo hacen. Para los flujos que admiten tiempos de espera, las propiedades ReadTimeout y WriteTimeout determinan el tiempo de espera deseado en milisegundos, donde 0 significa sin tiempo de espera. Los métodos Read y Write indican que se ha producido un tiempo de espera mediante la generación de una excepción.

Los métodos asincrónicos ReadAsync/WriteAsync no admiten tiempos de espera; en su lugar, puedes pasar un token de cancelación a estos métodos.

Seguridad de hilos (Thread Safety)

Por regla general, los flujos no son seguros para subprocesos, lo que significa que dos hilos no pueden leer o escribir concurrentemente en el mismo flujo sin posibles errores. La clase Stream ofrece una solución sencilla a través del método estático Synchronized. Este método acepta un flujo de cualquier tipo y devuelve un envoltorio seguro para subprocesos. El envoltorio funciona obteniendo un bloqueo exclusivo en cada operación de lectura, escritura o búsqueda, asegurando que solo un hilo pueda realizar dicha operación a la vez. En la práctica, esto permite que varios hilos añadan datos simultáneamente al mismo flujo; otros tipos de actividades (como la lectura simultánea) requieren bloqueos adicionales para garantizar que cada hilo acceda a la porción deseada del flujo. Discutimos la seguridad de hilos en detalle en el Capítulo 21.



A partir de .NET 6, puedes utilizar la clase RandomAccess para operaciones de E/S de archivos seguras para subprocesos y eficientes. RandomAccess también te permite pasar múltiples búferes para mejorar el rendimiento.

Flujos de Almacenamiento Subyacentes (Backing Store Streams)

La Figura 15-2 muestra los principales flujos de almacenamiento subyacentes proporcionados por .NET. También está disponible un "flujo nulo" a través del campo estático Null de la clase Stream. Los flujos nulos pueden ser útiles al escribir pruebas unitarias.

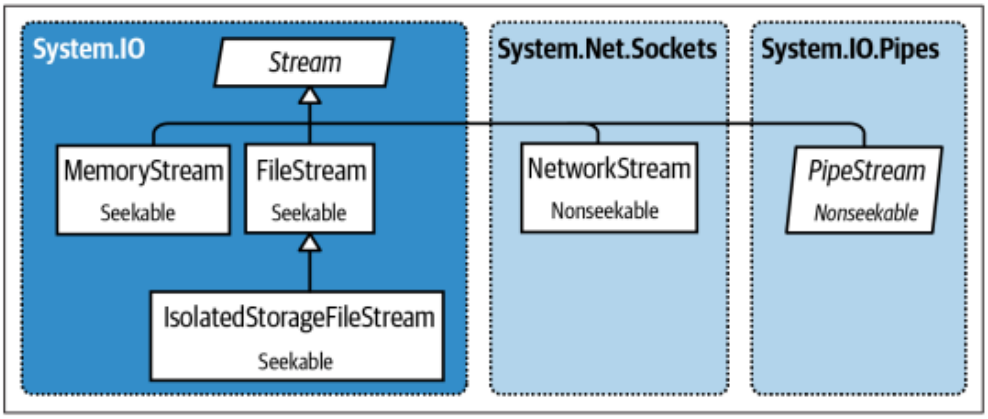


Figure 15-2. Backing store streams

En las siguientes secciones, describimos FileStream y MemoryStream; en la última sección de este capítulo, describimos IsolatedStorageStream. En el Capítulo 16, cubrimos NetworkStream.

FileStream

Anteriormente en esta sección, demostramos el uso básico de un FileStream para leer y escribir bytes de datos. Ahora examinaremos las características especiales de esta clase.


Si estás utilizando Universal Windows Platform (UWP), también puedes realizar operaciones de entrada/salida de archivos con los tipos en Windows.Storage. Describimos esto en el suplemento en línea en http://www.albahari.com/nutshell.

Construcción de un FileStream

La forma más sencilla de instanciar un FileStream es utilizando uno de los siguientes métodos de fachada estáticos de la clase File:

```
FileStream fs1 = File.OpenRead  ("readme.bin");          // Read-only
FileStream fs2 = File.OpenWrite ("writeme.tmp");         // Write-only
FileStream fs3 = File.Create    ("readwrite.tmp");       // Read/write
```

OpenWrite y Create difieren en su comportamiento si el archivo ya existe. Create trunca cualquier contenido existente; OpenWrite conserva el contenido existente con la posición del flujo en cero. Si escribes menos bytes de los que había previamente en el archivo, OpenWrite te dejará con una mezcla de contenido antiguo y nuevo.

También puedes instanciar directamente un FileStream. Sus constructores proporcionan acceso a todas las características, lo que te permite especificar un nombre de archivo o un identificador de archivo de bajo nivel, modos de creación y acceso de archivo, y opciones de uso compartido, almacenamiento en búfer y seguridad. El siguiente ejemplo abre un archivo existente para acceso de lectura/escritura sin sobrescribirlo (la palabra clave using asegura que se libere cuando fs sale del ámbito):

```
using var fs = new FileStream ("readwrite.tmp", FileMode.Open);
```

Examinares más de cerca FileMode en breve.

Especificar un nombre de archivo

Un nombre de archivo puede ser absoluto (por ejemplo, c:\temp\test.txt, o en Unix, /tmp/test.txt) o relativo al directorio actual (por ejemplo, test.txt o temp\test.txt). Puedes acceder o cambiar el directorio actual utilizando la propiedad estática Environment.CurrentDirectory.



Cuando un programa se inicia, el directorio actual puede o no coincidir con el directorio del ejecutable del programa. Por esta razón, nunca debes confiar en el directorio actual para localizar archivos adicionales empaquetados junto con tu ejecutable.

AppDomain.CurrentDomain.BaseDirectory devuelve el directorio base de la aplicación, que en casos normales es la carpeta que contiene el ejecutable del programa. Para especificar un nombre de archivo relativo a este directorio, puedes llamar a Path.Combine:

```
string baseFolder = AppDomain.CurrentDomain.BaseDirectory;
string logoPath = Path.Combine (baseFolder, "logo.jpg");
Console.WriteLine (File.Exists (logoPath));
```

Métodos Abreviados en la Clase File

En la clase File existen métodos estáticos que permiten leer o escribir un archivo completo de forma más sencilla. Los siguientes métodos estáticos leen un archivo completo en memoria de una sola vez:

- File.ReadAllText (devuelve una cadena de texto)
- File.ReadAllLines (devuelve un arreglo de cadenas de texto)
- File.ReadAllBytes (devuelve un arreglo de bytes)

Los siguientes métodos estáticos escriben un archivo completo de una sola vez:

- File.WriteAllText
- File.WriteAllLines

• File.WriteAllBytes

• File.AppendAllText (ideal para agregar contenido a un archivo de registro)

También existe un método estático llamado File.ReadLines, similar a ReadAllLines, pero devuelve un IEnumerable<string> que se evalúa de manera perezosa (lazy evaluation). Esto es más eficiente porque no carga todo el archivo en memoria de una sola vez. LINQ es ideal para consumir los resultados. Por ejemplo, el siguiente código calcula el número de líneas que tienen más de 80 caracteres de longitud:

```
int longLines = File.ReadLines ("filePath")
                    .Count (l => l.Length > 80);
```

Puedes leer y escribir a través de una red de Windows utilizando una ruta de Convención de Nombres Universal (UNC, por sus siglas en inglés), como \\JoesPC\PicShare\pic.jpg o \\10.1.1.2\PicShare\pic.jpg. (Para acceder a una carpeta compartida de Windows desde macOS o Unix, debes montarla en tu sistema de archivos siguiendo las instrucciones específicas de tu sistema operativo y luego abrirla utilizando una ruta normal desde C#).

Especificar un FileMode

Todos los constructores de FileStream que aceptan un nombre de archivo también requieren un argumento de tipo enumerado FileMode. La Figura 15-3 muestra cómo elegir un FileMode y las opciones ofrecen resultados similares a llamar a un método estático en la clase File.

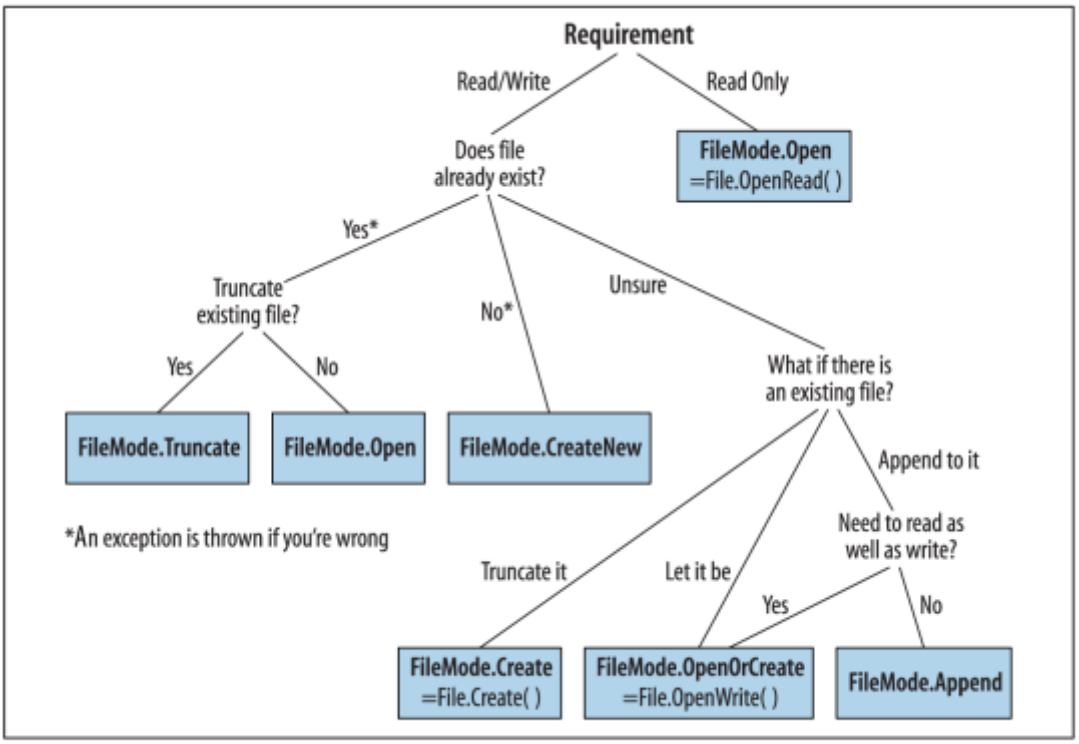


Figure 15-3. Choosing a FileMode


File.Create y FileMode.Create lanzarán una excepción si se utilizan en archivos ocultos. Para sobrescribir un archivo oculto, debes eliminarlo y volver a crearlo:

```
File.Delete ("hidden.txt");
using var file = File.Create ("hidden.txt");
...
```

Al construir un FileStream solo con un nombre de archivo y FileMode, obtienes (con una excepción) un flujo de lectura/escritura. Puedes solicitar una degradación si también proporcionas un argumento de tipo FileAccess:

```
[Flags]
public enum FileAccess { Read = 1, Write = 2, ReadWrite = 3 }
```

Lo siguiente devuelve un flujo de solo lectura, equivalente a llamar a File.OpenRead:

```
using var fs = new FileStream ("x.bin", FileMode.Open, FileAccess.Read);
...
```

FileMode.Append es el que se sale de lo común: con este modo, obtienes un flujo de solo escritura. Para realizar una operación de agregado (append) con soporte de lectura/escritura, debes usar FileMode.Open o FileMode.OpenOrCreate y luego buscar el final del flujo:

```
using var fs = new FileStream ("myFile.bin", FileMode.Open);

fs.Seek (0, SeekOrigin.End);
...
```

Características avanzadas de FileStream

Aquí hay otros argumentos opcionales que puedes incluir al construir un FileStream:

- Un enum FileShare que describe cuánto acceso otorgar a otros procesos que desean acceder al mismo archivo antes de que hayas terminado (None, Read [predeterminado], ReadWrite o Write).
- El tamaño, en bytes, del búfer interno (el valor predeterminado actualmente es de 4 KB).
- Un indicador booleano que indica si delegar en el sistema operativo para operaciones de E/S asíncronas.
- Un enum FileOptions para solicitar el cifrado del sistema operativo (Encrypted), eliminación automática al cerrar archivos temporales (DeleteOnClose) y sugerencias de optimización (RandomAccess y SequentialScan). También hay una bandera WriteThrough que solicita al sistema operativo deshabilitar el almacenamiento en caché de escritura diferida; esto es útil para archivos o registros transaccionales. Las banderas no compatibles con el sistema operativo subyacente se ignoran silenciosamente.

Al abrir un archivo con FileShare.ReadWrite, otros procesos o usuarios pueden leer y escribir simultáneamente en el mismo archivo. Para evitar problemas, todos pueden acordar bloquear secciones especificadas del archivo antes de leer o escribir, utilizando los siguientes métodos:

```
// Defined on the FileStream class:
public virtual void Lock   (long position, long length);
public virtual void Unlock (long position, long length);
```

Lock arroja una excepción si parte o toda la sección solicitada del archivo ya ha sido bloqueada.

MemoryStream

MemoryStream utiliza un arreglo como almacenamiento subyacente. Esto en parte contrarresta el propósito de tener un flujo, ya que todo el almacenamiento subyacente debe residir en memoria al mismo tiempo. MemoryStream sigue siendo útil cuando se necesita acceso aleatorio a un flujo no buscable. Si sabes que el flujo fuente será de un tamaño manejable, puedes copiarlo en un MemoryStream de la siguiente manera:

```
var ms = new MemoryStream();
sourceStream.CopyTo (ms);
```

Puedes convertir un MemoryStream en un arreglo de bytes llamando a ToArray. El método GetBuffer realiza la misma tarea de manera más eficiente al devolver una referencia directa al arreglo de almacenamiento subyacente; desafortunadamente, este arreglo suele ser más largo que la longitud real del flujo.



Cerrar y vaciar (flush) un MemoryStream es opcional. Si cierras un MemoryStream, ya no podrás leer ni escribir en él, pero aún se te permite llamar a ToArray para obtener los datos subyacentes. Flush no hace absolutamente nada en un MemoryStream.

Puedes encontrar más ejemplos de MemoryStream en "Compression Streams" en la página 695 y en "Overview" en la página 851.

PipeStream

PipeStream proporciona un medio sencillo mediante el cual un proceso puede comunicarse con otro a través del protocolo de tuberías del sistema operativo. Hay dos tipos de tuberías:

Tubería anónima (más rápida)

> Permite la comunicación unidireccional entre un proceso padre y un proceso hijo en la misma computadora.

Tubería con nombre (más flexible)

> Permite la comunicación bidireccional entre procesos arbitrarios en la misma computadora o en diferentes computadoras a través de una red.

Una tubería es útil para la comunicación entre procesos (IPC) en una sola computadora: no depende de un transporte de red, lo que significa que no hay sobrecarga de protocolo de red, y no tiene problemas con los firewalls.



Las tuberías se basan en flujos, por lo que un proceso espera recibir una serie de bytes mientras que otro proceso los envía. Una alternativa es que los procesos se comuniquen a través de un bloque de memoria compartida; describimos cómo hacer esto en "Memory-Mapped Files" en la página 712.

PipeStream es una clase abstracta con cuatro subtipos concretos. Dos se utilizan para tuberías anónimas y los otros dos para tuberías con nombre:

Tuberías anónimas

AnonymousPipeServerStream y AnonymousPipeClientStream.

Tuberías con nombre

NamedPipeServerStream y NamedPipeClientStream.

Las tuberías con nombre son más sencillas de usar, por lo que las describimos primero.

Tuberías con nombre

Con las tuberías con nombre, las partes se comunican a través de una tubería con el mismo nombre. El protocolo define dos roles distintos: el cliente y el servidor. La comunicación entre el cliente y el servidor ocurre de la siguiente manera:

- El servidor instancia un NamedPipeServerStream y luego llama a WaitForConnection.
- El cliente instancia un NamedPipeClientStream y luego llama a Connect (con un tiempo de espera opcional).

Las dos partes luego leen y escriben en los flujos para comunicarse.

El siguiente ejemplo muestra un servidor que envía un solo byte (100) y luego espera recibir un solo byte:

```
using var s = new NamedPipeServerStream ("pipedream");

s.WaitForConnection();
s.WriteByte (100);                // Send the value 100.
Console.WriteLine (s.ReadByte());
```

Aquí está el código correspondiente del cliente:

```
using var s = new NamedPipeClientStream ("pipedream");

s.Connect();
Console.WriteLine (s.ReadByte());
s.WriteByte (200);                 // Send the value 200 back.
```

Los flujos de tubería con nombre son bidireccionales de forma predeterminada, por lo que cualquiera de las partes puede leer o escribir en su flujo. Esto significa que el cliente y el servidor deben acordar algún protocolo para coordinar sus acciones, de modo que ambas partes no

terminen enviando o recibiendo al mismo tiempo. También debe haber un acuerdo sobre la longitud de cada transmisión. Nuestro ejemplo fue trivial en este sentido, porque solo enviamos un solo byte en cada dirección.

Para ayudar con mensajes más largos que un solo byte, las tuberías proporcionan un modo de transmisión de mensajes (solo en Windows). Si esto está habilitado, una parte que llama a Read puede saber cuándo se completa un mensaje comprobando la propiedad IsMessageComplete. Para demostrarlo, comenzamos escribiendo un método auxiliar que lee un mensaje completo desde un PipeStream habilitado para mensajes, es decir, lee hasta que IsMessageComplete es true:

```
static byte[] ReadMessage (PipeStream s)
{
  MemoryStream ms = new MemoryStream();
  byte[] buffer = new byte [0x1000];      // Read in 4 KB blocks

  do    { ms.Write (buffer, 0, s.Read (buffer, 0, buffer.Length)); }
  while (!s.IsMessageComplete);

  return ms.ToArray();
}
```

Para hacer esto de forma asíncrona, reemplace "s.Read" con "await s.ReadAsync".



No se puede determinar si un PipeStream ha terminado de leer un mensaje simplemente esperando a que Read devuelva 0. Esto se debe a que, a diferencia de la mayoría de los otros tipos de flujos, los flujos de tubería y los flujos de red no tienen un final definido. En su lugar, se "secan" temporalmente entre las transmisiones de mensajes.

Ahora podemos activar el modo de transmisión de mensajes. En el servidor, esto se hace especificando PipeTransmissionMode.Message al construir el flujo:

```
using var s = new NamedPipeServerStream ("pipedream", PipeDirection.InOut,
                                         1, PipeTransmissionMode.Message);

s.WaitForConnection();

byte[] msg = Encoding.UTF8.GetBytes ("Hello");
s.Write (msg, 0, msg.Length);

Console.WriteLine (Encoding.UTF8.GetString (ReadMessage (s)));
```

En el cliente, activamos el modo de transmisión de mensajes configurando ReadMode después de llamar a Connect:

```
using var s = new NamedPipeClientStream ("pipedream");

s.Connect();
s.ReadMode = PipeTransmissionMode.Message;

Console.WriteLine (Encoding.UTF8.GetString (ReadMessage (s)));

byte[] msg = Encoding.UTF8.GetBytes ("Hello right back!");
s.Write (msg, 0, msg.Length);
```



El modo de mensaje solo es compatible con Windows. En otras plataformas, se lanzará una excepción PlatformNotSupportedException.

Tuberías anónimas

Una tubería anónima proporciona un flujo de comunicación unidireccional entre un proceso padre y un proceso hijo. En lugar de usar un nombre válido en todo el sistema, las tuberías anónimas se conectan a través de un identificador privado.

Al igual que con las tuberías con nombre, hay roles distintos para el cliente y el servidor. Sin embargo, el sistema de comunicación es un poco diferente y sigue estos pasos:

1. El servidor instancia un AnonymousPipeServerStream, comprometiéndose a usar una dirección de tubería de entrada (In) o salida (Out).

2. El servidor llama a GetClientHandleAsString para obtener un identificador de la tubería, que luego pasa al cliente (normalmente como argumento al iniciar el proceso hijo).

3. El proceso hijo instancia un AnonymousPipeClientStream, especificando la dirección de la tubería opuesta.

4. El servidor libera el identificador local que se generó en el paso 2, llamando a DisposeLocalCopyOfClientHandle.

5. Los procesos padre e hijo se comunican leyendo/ escribiendo en el flujo.

Debido a que las tuberías anónimas son unidireccionales, un servidor debe crear dos tuberías para la comunicación bidireccional. El siguiente programa de consola crea dos tuberías (entrada y salida) y luego inicia un proceso hijo. Luego envía un solo byte al proceso hijo y recibe un solo byte a cambio:

```
class Program
{
  static void Main (string[] args)
  {
    if (args.Length == 0)
      // No arguments signals server mode
      AnonymousPipeServer();
    else
      // We pass in the pipe handle IDs as arguments to signal client mode
      AnonymousPipeClient (args [0], args [1]);
  }

  static void AnonymousPipeClient (string rxID, string txID)
  {
    using (var rx = new AnonymousPipeClientStream (PipeDirection.In, rxID))
    using (var tx = new AnonymousPipeClientStream (PipeDirection.Out, txID))
    {
      Console.WriteLine ("Client received: " + rx.ReadByte ());
      tx.WriteByte (200);
    }
  }

  static void AnonymousPipeServer ()
  {
    using var tx = new AnonymousPipeServerStream (
                     PipeDirection.Out, HandleInheritability.Inheritable);
    using var rx = new AnonymousPipeServerStream (
                     PipeDirection.In, HandleInheritability.Inheritable);

    string txID = tx.GetClientHandleAsString ();
    string rxID = rx.GetClientHandleAsString ();

    // Create and start up a child process.
    // We'll use the same Console executable, but pass in arguments:
    string thisAssembly = Assembly.GetEntryAssembly().Location;
    string thisExe = Path.ChangeExtension (thisAssembly, ".exe");
    var args = $"{txID} {rxID}";
    var startInfo = new ProcessStartInfo (thisExe, args);

    startInfo.UseShellExecute = false;       // Required for child process
    Process p = Process.Start (startInfo);


    tx.DisposeLocalCopyOfClientHandle ();    // Release unmanaged
    rx.DisposeLocalCopyOfClientHandle ();    // handle resources.

    tx.WriteByte (100);    // Send a byte to the child process

    Console.WriteLine ("Server received: " + rx.ReadByte ());

    p.WaitForExit ();
  }
}
```

Al igual que con las tuberías con nombre, el cliente y el servidor deben coordinar el envío y recepción de datos, y acordar la longitud de cada transmisión. Desafortunadamente, las tuberías anónimas no admiten el modo de mensaje, por lo que debes implementar tu propio protocolo para acordar la longitud de los mensajes. Una solución es enviar, en los primeros cuatro bytes de cada transmisión, un valor entero que defina la longitud del mensaje que sigue. La clase BitConverter proporciona métodos para convertir entre un entero y un array de cuatro bytes.

Flujo con búfer (BufferedStream)

BufferedStream es una clase que envuelve o encapsula otro flujo (stream) con capacidad de almacenamiento en búfer, y es uno de varios tipos de flujos decoradores en .NET, que se ilustran en la Figura 15-4.

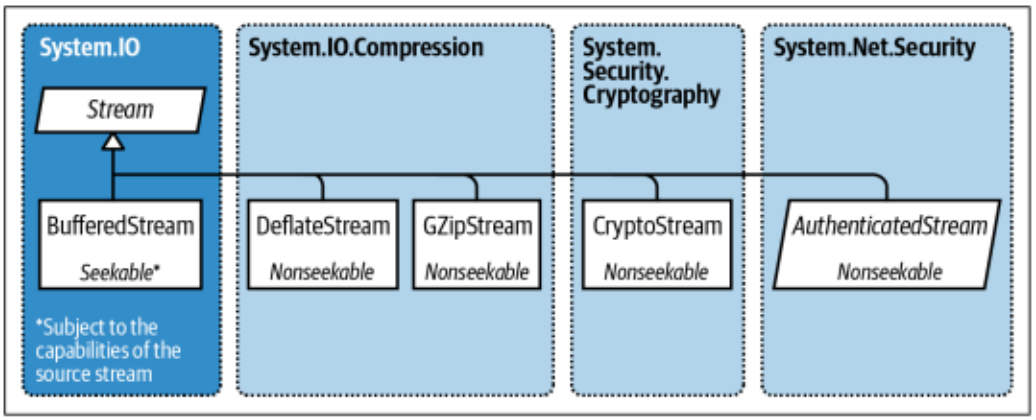


Figure 15-4. Decorator streams

El almacenamiento en búfer mejora el rendimiento al reducir las idas y vueltas al almacén de respaldo. Aquí te mostramos cómo envolvemos un FileStream en un BufferedStream de 20 KB:

```
// Write 100K to a file:
File.WriteAllBytes ("myFile.bin", new byte [100000]);

using FileStream fs = File.OpenRead ("myFile.bin");
using BufferedStream bs = new BufferedStream (fs, 20000);  //20K buffer

bs.ReadByte();
Console.WriteLine (fs.Position);         // 20000
```

En este ejemplo, el flujo subyacente avanza 20.000 bytes después de leer solo un byte, gracias al almacenamiento en búfer anticipado. Podríamos llamar a ReadByte otras 19.999 veces antes de que se acceda nuevamente al FileStream.

Acoplar un BufferedStream a un FileStream, como en este ejemplo, tiene un valor limitado porque FileStream ya tiene un almacenamiento en búfer incorporado. Su único uso podría ser para ampliar el búfer en un FileStream ya construido.

Al cerrar un BufferedStream, se cierra automáticamente el flujo de respaldo subyacente.

Adaptadores de flujo

Un Stream trabaja únicamente con bytes; para leer o escribir tipos de datos como cadenas, enteros o elementos XML, debes usar un adaptador. Esto es lo que proporciona .NET:

Adaptadores de texto (para datos de cadena y caracteres)

```
TextReader, TextWriter
StreamReader, StreamWriter
StringReader, StringWriter
```

Adaptadores binarios (para tipos primitivos como int, bool, string y float)

`BinaryReader, BinaryWriter`

Adaptadores XML (cubiertos en el Capítulo 11)

`XmlReader, XmlWriter`

La Figura 15-5 ilustra las relaciones entre estos tipos.

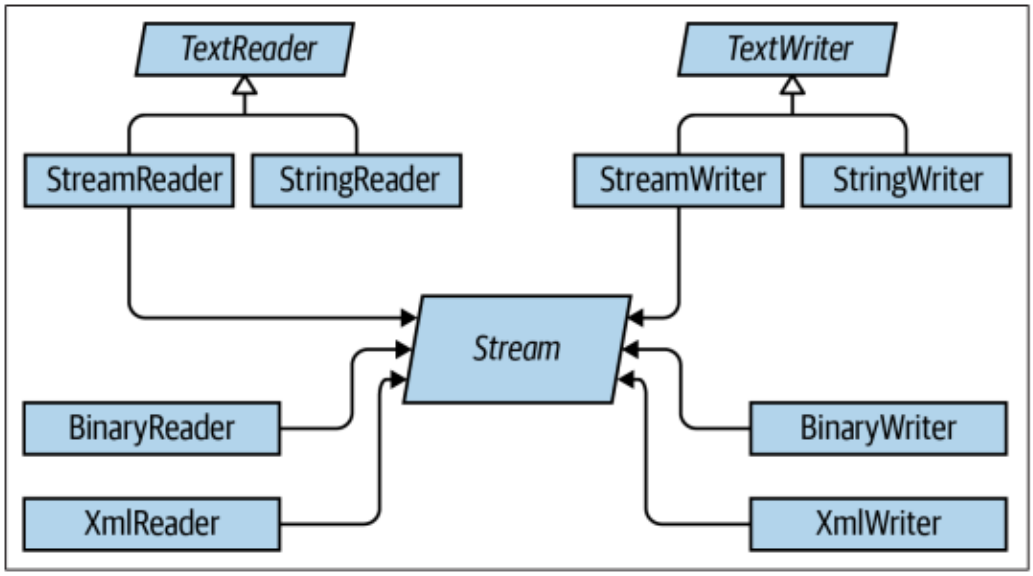


Figure 15-5. Readers and writers

Los adaptadores de texto

TextReader y TextWriter son las clases base abstractas para los adaptadores que tratan exclusivamente con caracteres y cadenas. Cada una tiene dos implementaciones de propósito general en .NET:

StreamReader/StreamWriter

> Utiliza un Stream como su almacén de datos en bruto, traduciendo los bytes del stream en caracteres o cadenas.

StringReader/StringWriter

> Implementa TextReader/TextWriter utilizando cadenas en memoria.

La Tabla 15-2 enumera los miembros de TextReader por categoría. Peek devuelve el siguiente carácter en el flujo sin avanzar la posición. Tanto Peek como la versión sin argumentos de Read devuelven -1 si se encuentra al final del flujo; de lo contrario, devuelven un entero que se puede convertir directamente a un char. La sobrecarga de Read que acepta un buffer char[] es idéntica en funcionalidad al método ReadBlock. ReadLine lee hasta encontrar un CR (carácter 13) o LF (carácter 10), o una combinación CR+LF en secuencia. Luego devuelve una cadena, descartando los caracteres CR/LF.

Table 15-2. TextReader members

Category	Members
Reading one char	`public virtual int Peek(); // Cast the result to a char`
	`public virtual int Read(); // Cast the result to a char`
Reading many chars	`public virtual int Read (char[] buffer, int index, int count);`
	`public virtual int ReadBlock (char[] buffer, int index, int count);`
	`public virtual string ReadLine();`
	`public virtual string ReadToEnd();`
Closing	`public virtual void Close();`
	`public void Dispose(); // Same as Close`
Other	`public static readonly TextReader Null;`
	`public static TextReader Synchronized (TextReader reader);`



Environment.NewLine devuelve la secuencia de nueva línea para el sistema operativo actual.

En Windows, esto es "\r\n" (piensa en "ReturN") y se inspira vagamente en una máquina de escribir mecánica: un CR (carácter 13) seguido de un LF (carácter 10). ¡Si inviertes el orden, obtendrás dos nuevas líneas o ninguna!

En Unix y macOS, simplemente es "\n".

TextWriter tiene métodos análogos para escribir, como se muestra en la Tabla 15-3. Los métodos Write y WriteLine también se sobrecargan para aceptar todos los tipos primitivos, además del tipo object. Estos métodos simplemente llaman al método ToString en lo que se pasa (opcionalmente a través de un IFormatProvider especificado ya sea al llamar al método o al construir el TextWriter).

Table 15-3. TextWriter members

Category	Members
Writing one char	`public virtual void Write (char value);`
Writing many chars	`public virtual void Write (string value);`
	`public virtual void Write (char[] buffer, int index, int count);`
	`public virtual void Write (string format, params object[] arg);`
	`public virtual void WriteLine (string value);`
Closing and flushing	`public virtual void Close();`
	`public void Dispose(); // Same as Close`
	`public virtual void Flush();`
Formatting and encoding	`public virtual IFormatProvider FormatProvider { get; }`
	`public virtual string NewLine { get; set; }`
	`public abstract Encoding Encoding { get; }`
Other	`public static readonly TextWriter Null;`
	`public static TextWriter Synchronized (TextWriter writer);`

WriteLine simplemente agrega el texto dado con Environment.NewLine. Puedes cambiar esto a través de la propiedad NewLine (esto puede ser útil para la interoperabilidad con formatos de archivo de Unix).



Al igual que Stream, TextReader y TextWriter ofrecen versiones asincrónicas basadas en tareas de sus métodos de lectura/escritura.

Flujos y E/SStreamReader y StreamWriter

En el siguiente ejemplo, un StreamWriter escribe dos líneas de texto en un archivo, y luego un StreamReader lee el archivo de vuelta:

```
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
{
  writer.WriteLine ("Line1");
  writer.WriteLine ("Line2");
}

using (FileStream fs = File.OpenRead ("test.txt"))
using (TextReader reader = new StreamReader (fs))
{
  Console.WriteLine (reader.ReadLine());       // Line1
  Console.WriteLine (reader.ReadLine());       // Line2
}
```

Dado que los adaptadores de texto están frecuentemente relacionados con archivos, la clase File proporciona los métodos estáticos CreateText, AppendText y OpenText para simplificar el proceso:

```
using (TextWriter writer = File.CreateText ("test.txt"))
{
  writer.WriteLine ("Line1");
  writer.WriteLine ("Line2");
}

using (TextWriter writer = File.AppendText ("test.txt"))
  writer.WriteLine ("Line3");

using (TextReader reader = File.OpenText ("test.txt"))
  while (reader.Peek() > -1)
    Console.WriteLine (reader.ReadLine());      // Line1
                                                // Line2
                                                // Line3
```

Esto también ilustra cómo comprobar el final de un archivo (es decir, reader.Peek()). Otra opción es leer hasta que reader.ReadLine devuelva null.

También puedes leer y escribir otros tipos, como enteros, pero debido a que TextWriter invoca ToString en tu tipo, debes analizar una cadena al leerla de nuevo:

```
using (TextWriter w = File.CreateText ("data.txt"))
{
  w.WriteLine (123);          // Writes "123"
  w.WriteLine (true);         // Writes the word "true"
}

using (TextReader r = File.OpenText ("data.txt"))
{
  int myInt = int.Parse (r.ReadLine());     // myInt == 123
  bool yes = bool.Parse (r.ReadLine());     // yes == true
}
```

Los codificadores de caracteres

TextReader y TextWriter son en sí mismos solo clases abstractas sin conexión a un flujo o almacenamiento de respaldo. Sin embargo, los tipos StreamReader y StreamWriter están conectados a un flujo orientado a bytes subyacente, por lo que deben convertir entre caracteres y bytes. Lo hacen a través de una clase de codificación (Encoding) del espacio de nombres System.Text, que eliges al construir el StreamReader o StreamWriter. Si no eliges ninguno, se utiliza la codificación predeterminada UTF-8.



Si especificas explícitamente una codificación, StreamWriter, de forma predeterminada, escribirá un prefijo al inicio del flujo para identificar la codificación. Esto generalmente no es deseado, y puedes evitarlo construyendo la codificación de la siguiente manera:

```
var encoding = new UTF8Encoding (
  encoderShouldEmitUTF8Identifier:false,
  throwOnInvalidBytes:true);
```

El segundo argumento indica al StreamWriter (o StreamReader) que genere una excepción si encuentra bytes que no tienen una traducción de cadena válida para su codificación. Esto coincide con su comportamiento predeterminado si no especificas una codificación.

El codificador más simple es ASCII porque cada carácter se representa con un byte. La codificación ASCII mapea los primeros 127 caracteres del conjunto Unicode en su único byte, abarcando lo que se ve en un teclado de estilo estadounidense. La mayoría de los demás caracteres, incluidos los símbolos especializados y los caracteres no ingleses, no se pueden representar y se convierten en el carácter □. La codificación UTF-8 predeterminada puede mapear todos los caracteres Unicode asignados, pero es más compleja. Los primeros 127 caracteres se codifican en un solo byte, para compatibilidad con ASCII; los caracteres restantes se codifican en un número variable de bytes (comúnmente dos o tres). Considera lo siguiente:

```
using (TextWriter w = File.CreateText ("but.txt"))    // Use default UTF-8
  w.WriteLine ("but-");                               // encoding.

using (Stream s = File.OpenRead ("but.txt"))
  for (int b; (b = s.ReadByte()) > -1;)
    Console.WriteLine (b);
```

La palabra "but" está seguida no por un guion estándar, sino por el carácter de raya larga (—), U+2014. ¡Este es el que no te causará problemas con tu editor de libros! Veamos la salida:

```
98     // b
117    // u
116    // t
226    // em dash byte 1       Note that the byte values
128    // em dash byte 2       are >= 128 for each part
148    // em dash byte 3       of the multibyte sequence.
13     // <CR>
10     // <LF>
```

Debido a que la raya larga está fuera de los primeros 127 caracteres del conjunto Unicode, requiere más de un solo byte para codificarse en UTF-8 (en este caso, tres). UTF-8 es eficiente con el alfabeto occidental, ya que la mayoría de los caracteres populares consumen solo un byte. También se puede degradar fácilmente a ASCII simplemente ignorando todos los bytes por encima de 127. Su desventaja es que buscar dentro de un flujo es problemático porque la posición de un carácter no se corresponde con su posición de byte en el flujo. Una alternativa es UTF-16 (etiquetado simplemente como "Unicode" en la clase Encoding). Así es como escribimos la misma cadena con UTF-16:

```
using (Stream s = File.Create ("but.txt"))
using (TextWriter w = new StreamWriter (s, Encoding.Unicode))
  w.WriteLine ("but-");

foreach (byte b in File.ReadAllBytes ("but.txt"))
  Console.WriteLine (b);
```

Y aquí está la salida:

```
255    // Byte-order mark 1
254    // Byte-order mark 2
98     // 'b' byte 1
0      // 'b' byte 2
117    // 'u' byte 1
0      // 'u' byte 2
116    // 't' byte 1
0      // 't' byte 2
20     // '--' byte 1
32     // '--' byte 2
13     // <CR> byte 1
0      // <CR> byte 2
10     // <LF> byte 1
0      // <LF> byte 2
```

Técnicamente, UTF-16 utiliza dos o cuatro bytes por carácter (hay cerca de un millón de caracteres Unicode asignados o reservados, por lo que dos bytes no siempre son suficientes). Sin embargo, debido a que el tipo char de C# solo tiene 16 bits de ancho, una codificación UTF-16 siempre usará exactamente dos bytes por char de .NET. Esto facilita saltar a un índice de carácter específico dentro de un flujo.

UTF-16 utiliza un prefijo de dos bytes para identificar si los pares de bytes se escriben en orden "little-endian" o "big-endian" (el byte menos significativo primero o el byte más significativo primero). El orden little-endian predeterminado es estándar para los sistemas basados en Windows.

StringReader y StringWriter

Los adaptadores StringReader y StringWriter no envuelven un flujo en absoluto; en cambio, utilizan una cadena o un StringBuilder como origen de datos subyacente. Esto significa que no se requiere ninguna traducción de bytes, de hecho, las clases no hacen nada que no se pueda lograr fácilmente con una cadena o un StringBuilder junto con una variable de índice. Sin embargo, su ventaja es que comparten una clase base con StreamReader/StreamWriter.

Por ejemplo, supongamos que tenemos una cadena que contiene XML y queremos analizarla con un XmlReader. El método XmlReader.Create acepta uno de los siguientes:

- Una URI
- Un flujo (Stream)
- Un TextReader

Entonces, ¿cómo analizamos XML a partir de nuestra cadena? Debido a que StringReader es una subclase de TextReader, tenemos suerte. Podemos instanciar y pasar un StringReader de la siguiente manera:

```
XmlReader r = XmlReader.Create (new StringReader (myString));
```

Adaptadores binarios

BinaryReader y BinaryWriter leen y escriben tipos de datos nativos: bool, byte, char, decimal, float, double, short, int, long, sbyte, ushort, uint y ulong, así como cadenas y matrices de los tipos de datos primitivos.

A diferencia de StreamReader y StreamWriter, los adaptadores binarios almacenan los tipos de datos primitivos de manera eficiente porque se representan en memoria. Por lo tanto, un int usa cuatro bytes y un double usa ocho bytes. Las cadenas se escriben a través de una codificación de texto (como con StreamReader y StreamWriter), pero se les agrega un prefijo de longitud para poder leer una serie de cadenas sin necesidad de delimitadores especiales.

Imaginemos que tenemos un tipo simple definido de la siguiente manera:

```
public class Person
{
  public string Name;
  public int    Age;
  public double Height;
}
```

Podemos agregar los siguientes métodos a Persona para guardar/cargar sus datos en/desde un flujo utilizando adaptadores binarios:

```
public void SaveData (Stream s)
{
  var w = new BinaryWriter (s);
  w.Write (Name);
  w.Write (Age);
  w.Write (Height);
  w.Flush();         // Ensure the BinaryWriter buffer is cleared.
                     // We won't dispose/close it, so more data
}                    // can be written to the stream.

public void LoadData (Stream s)
{
  var r = new BinaryReader (s);
  Name   = r.ReadString();
  Age    = r.ReadInt32();
  Height = r.ReadDouble();
}
```

BinaryReader también puede leer en matrices de bytes. Lo siguiente lee todo el contenido de un flujo que se puede buscar (seekable stream):

```
byte[] data = new BinaryReader (s).ReadBytes ((int) s.Length);
```

Esto es más conveniente que leer directamente del flujo porque no requiere un bucle para asegurarse de que se haya leído todos los datos.

Cierre y liberación de recursos de los adaptadores de flujo

Tienes cuatro opciones para finalizar los adaptadores de flujo:

1. Cerrar solo el adaptador.

2. Cerrar el adaptador y luego cerrar el flujo.

3. (Para escritores) Hacer un volcado (flush) del adaptador y luego cerrar el flujo.

4. (Para lectores) Cerrar solo el flujo.

 Close y Dispose son sinónimos en los adaptadores, al igual que en los flujos.

Las opciones 1 y 2 son semánticamente idénticas porque cerrar un adaptador automáticamente cierra el flujo subyacente. Cuando anidas declaraciones using, estás implícitamente optando por la opción 2:

```
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
  writer.WriteLine ("Line");
```

Debido a que el cierre se realiza desde el interior hacia el exterior, el adaptador se cierra primero y luego el flujo. Además, si se produce una excepción dentro del constructor del adaptador, el flujo se cerrará igualmente. ¡Es difícil equivocarse con las declaraciones using anidadas!

 Nunca cierres un flujo antes de cerrar o hacer un volcado (flush) de su escritor; podrías perder cualquier dato que esté almacenado en el búfer del adaptador.

Las opciones 3 y 4 funcionan porque los adaptadores están en la categoría inusual de objetos desechables opcionalmente. Un ejemplo de cuándo podrías elegir no desechar un adaptador es cuando has terminado con el adaptador pero quieres dejar el flujo subyacente abierto para su uso posterior:

```
using (FileStream fs = new FileStream ("test.txt", FileMode.Create))
{
  StreamWriter writer = new StreamWriter (fs);
  writer.WriteLine ("Hello");
  writer.Flush();

  fs.Position = 0;


 Console.WriteLine (fs.ReadByte());
}
```

Aquí, escribimos en un archivo, reposicionamos el flujo y luego leemos el primer byte antes de cerrar el flujo. Si desecháramos el StreamWriter, también cerraría el FileStream subyacente, lo que haría que la lectura posterior fallara. La condición es que llamemos a Flush para asegurarnos de que el búfer del StreamWriter se escriba en el flujo subyacente.



Los adaptadores de flujo, con su semántica de eliminación opcional, no implementan el patrón extendido de eliminación en el que el finalizador llama a Dispose. Esto permite que un adaptador abandonado evite la eliminación automática cuando el recolector de basura lo alcanza.

También hay un constructor en StreamReader/StreamWriter que le indica que mantenga el flujo abierto después de la eliminación. En consecuencia, podemos reescribir el ejemplo anterior de la siguiente manera:

```
using (var fs = new FileStream ("test.txt", FileMode.Create))
{
  using (var writer = new StreamWriter (fs, new UTF8Encoding (false, true),
                                         0x400, true))
    writer.WriteLine ("Hello");

  fs.Position = 0;
  Console.WriteLine (fs.ReadByte());
  Console.WriteLine (fs.Length);
}
```

Flujos de compresión

Se proporcionan dos flujos de compresión de uso general en el espacio de nombres System.IO.Compression: DeflateStream y GZipStream. Ambos utilizan un algoritmo de compresión popular similar al formato ZIP. La diferencia es que GZipStream escribe un protocolo adicional al inicio y al final, incluyendo un CRC para detectar errores. GZipStream también cumple con un estándar reconocido por otros software.

.NET también incluye BrotliStream, que implementa el algoritmo de compresión Brotli. BrotliStream es más de 10 veces más lento que DeflateStream y GZipStream, pero logra una mejor relación de compresión. (El impacto en el rendimiento solo se aplica a la compresión; la descompresión funciona muy bien).

Los tres flujos permiten lectura y escritura, con las siguientes consideraciones:

- Siempre se escribe en el flujo al comprimir.
- Siempre se lee del flujo al descomprimir.

DeflateStream, GZipStream y BrotliStream son decoradores; comprimen o descomprimen datos de otro flujo que se proporciona en la construcción. En el siguiente ejemplo, comprimimos y descomprimimos una serie de bytes utilizando un FileStream como almacenamiento subyacente:

```
using (Stream s = File.Create ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Compress))
  for (byte i = 0; i < 100; i++)
    ds.WriteByte (i);

using (Stream s = File.OpenRead ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Decompress))
  for (byte i = 0; i < 100; i++)
    Console.WriteLine (ds.ReadByte());     // Writes 0 to 99
```

Con DeflateStream, el archivo comprimido ocupa 102 bytes; ligeramente más grande que el original (BrotliStream lo comprimiría a 73 bytes). La compresión funciona mal con datos binarios "densos" y no repetitivos (y aún peor con datos encriptados, que carecen de regularidad por diseño). Sin embargo, funciona bien con la mayoría de los archivos de texto.

En el siguiente ejemplo, comprimimos y descomprimimos un flujo de texto compuesto por 1,000 palabras elegidas al azar de una pequeña frase utilizando el algoritmo Brotli. Esto también demuestra el encadenamiento de un flujo de almacenamiento subyacente, un flujo decorador y un adaptador (como se muestra al principio del capítulo en la Figura 15-1), y el uso de métodos asíncronos:

```
string[] words = "The quick brown fox jumps over the lazy dog".Split();
Random rand = new Random (0);   // Give it a seed for consistency

using (Stream s = File.Create ("compressed.bin"))
using (Stream ds = new BrotliStream (s, CompressionMode.Compress))
using (TextWriter w = new StreamWriter (ds))
  for (int i = 0; i < 1000; i++)
    await w.WriteAsync (words [rand.Next (words.Length)] + " ");

Console.WriteLine (new FileInfo ("compressed.bin").Length);      // 808

using (Stream s = File.OpenRead ("compressed.bin"))
using (Stream ds = new BrotliStream (s, CompressionMode.Decompress))
using (TextReader r = new StreamReader (ds))
  Console.Write (await r.ReadToEndAsync());  // Output below:

lazy lazy the fox the quick The brown fox jumps over fox over fox The
brown brown brown over brown quick fox brown dog dog lazy fox dog brown
over fox jumps lazy lazy quick The jumps fox jumps The over jumps dog...
```

En este caso, BrotliStream comprime eficientemente a 808 bytes, menos de un byte por palabra. (Para comparación, DeflateStream comprime los mismos datos a 885 bytes).

Compresión en memoria

A veces, necesitas comprimir completamente en memoria. Aquí tienes cómo usar un MemoryStream con este propósito:

```
byte[] data = new byte[1000];          // We can expect a good compression
                                       // ratio from an empty array!
var ms = new MemoryStream();
using (Stream ds = new DeflateStream (ms, CompressionMode.Compress))
  ds.Write (data, 0, data.Length);

byte[] compressed = ms.ToArray();
Console.WriteLine (compressed.Length);       // 11

// Decompress back to the data array:
ms = new MemoryStream (compressed);
using (Stream ds = new DeflateStream (ms, CompressionMode.Decompress))
  for (int i = 0; i < 1000; i += ds.Read (data, i, 1000 - i));
```

La instrucción `using` alrededor de `DeflateStream` lo cierra de manera convencional, lo que implica que también se cierra el `MemoryStream` que envuelve. Esto significa que debemos llamar al método `ToArray` para extraer los datos.

Aquí tienes una alternativa que evita cerrar el `MemoryStream` y utiliza los métodos asíncronos de lectura y escritura:

```
byte[] data = new byte[1000];

MemoryStream ms = new MemoryStream();
using (Stream ds = new DeflateStream (ms, CompressionMode.Compress, true))
  await ds.WriteAsync (data, 0, data.Length);

Console.WriteLine (ms.Length);             // 113
ms.Position = 0;
using (Stream ds = new DeflateStream (ms, CompressionMode.Decompress))
  for (int i = 0; i < 1000; i += await ds.ReadAsync (data, i, 1000 - i));
```

El indicador adicional que se envía al constructor de `DeflateStream` le indica que no siga el protocolo habitual de llevar consigo el `MemoryStream` subyacente al ser eliminado. En otras palabras, el `MemoryStream` se mantiene abierto, lo que nos permite posicionarlo nuevamente en cero y volver a leerlo.

Compresión de archivos en formato Unix gzip

El algoritmo de compresión de `GZipStream` es popular en sistemas Unix como formato de compresión de archivos. Cada archivo fuente se comprime en un archivo de destino separado con la extensión `.gz`.

Los siguientes métodos realizan el trabajo de las utilidades de línea de comandos gzip y gunzip en Unix:

```
async Task GZip (string sourcefile, bool deleteSource = true)
{
  var gzipfile = $"{sourcefile}.gz";
  if (File.Exists (gzipfile))
    throw new Exception ("Gzip file already exists");

  // Compress
  using (FileStream inStream = File.Open (sourcefile, FileMode.Open))
  using (FileStream outStream = new FileStream (gzipfile, FileMode.CreateNew))
```

```
  using (GZipStream gzipStream =
    new GZipStream (outStream, CompressionMode.Compress))
    await inStream.CopyToAsync (gzipStream);

  if (deleteSource) File.Delete(sourcefile);
}

async Task GUnzip (string gzipfile, bool deleteGzip = true)
{
  if (Path.GetExtension (gzipfile) != ".gz")
    throw new Exception ("Not a gzip file");

  var uncompressedFile = gzipfile.Substring (0, gzipfile.Length - 3);
  if (File.Exists (uncompressedFile))
    throw new Exception ("Destination file already exists");

  // Uncompress
  using (FileStream uncompressToStream =
         File.Open (uncompressedFile, FileMode.Create))
  using (FileStream zipfileStream = File.Open (gzipfile, FileMode.Open))
  using (var unzipStream =
         new GZipStream (zipfileStream, CompressionMode.Decompress))
    await unzipStream.CopyToAsync (uncompressToStream);

  if (deleteGzip) File.Delete (gzipfile);
}
```

La siguiente instrucción comprime un archivo:

```
await GZip ("/tmp/myfile.txt");      // Creates /tmp/myfile.txt.gz
```

Y la siguiente instrucción descomprime el archivo:

```
await GUnzip ("/tmp/myfile.txt.gz")  // Creates /tmp/myfile.txt
```

Trabajando con archivos ZIP

Las clases ZipArchive y ZipFile en System.IO.Compression admiten el formato de compresión ZIP. La ventaja del formato ZIP sobre DeflateStream y GZipStream es que actúa como un contenedor para múltiples archivos y es compatible con archivos ZIP creados con Windows Explorer.



ZipArchive y ZipFile funcionan tanto en Windows como en Unix; sin embargo, el formato es más popular en Windows. En Unix, el formato .tar es más popular como contenedor para múltiples archivos. Puedes leer/escribir archivos .tar utilizando una biblioteca de terceros como SharpZipLib.

ZipArchive funciona con flujos (streams), mientras que ZipFile aborda el escenario más común de trabajar con archivos (es una clase de ayuda estática para ZipArchive).

El método CreateFromDirectory de ZipFile agrega todos los archivos de un directorio especificado a un archivo ZIP:

```
ZipFile.CreateFromDirectory (@"d:\MyFolder", @"d:\archive.zip");
```

El método `ExtractToDirectory` hace lo contrario y extrae un archivo ZIP a un directorio:

```
ZipFile.ExtractToDirectory (@"d:\archive.zip", @"d:\MyFolder");
```

Cuando comprimes, puedes especificar si deseas optimizar el tamaño del archivo o la velocidad, así como si incluir o no el nombre del directorio fuente en el archivo. Si habilitas esta última opción en nuestro ejemplo, se creará un subdirectorio en el archivo ZIP llamado "MyFolder" donde se colocarán los archivos comprimidos.

`ZipFile` tiene un método `Open` para leer/escribir entradas individuales. Esto devuelve un objeto `ZipArchive` (que también puedes obtener mediante la instanciación de `ZipArchive` con un objeto `Stream`). Al llamar a `Open`, debes especificar un nombre de archivo e indicar si deseas leer, crear o actualizar el archivo. Luego, puedes enumerar las entradas existentes a través de la propiedad `Entries` o encontrar un archivo en particular llamando a `GetEntry`:

```
using (ZipArchive zip = ZipFile.Open (@"d:\zz.zip", ZipArchiveMode.Read))

  foreach (ZipArchiveEntry entry in zip.Entries)
    Console.WriteLine (entry.FullName + " " + entry.Length);
```

`ZipArchiveEntry` también tiene un método `Delete`, un método `ExtractToFile` (que en realidad es un método de extensión en la clase `ZipFileExtensions`) y un método `Open` que devuelve un `Stream` legible y escribible. Puedes crear nuevas entradas llamando a `CreateEntry` (o al método de extensión `CreateEntryFromFile`) en el `ZipArchive`.

El siguiente código crea el archivo `d:\zz.zip`, al cual agrega `foo.dll` bajo una estructura de directorios dentro del archivo llamada `bin\X86`:

```
byte[] data = File.ReadAllBytes (@"d:\foo.dll");
using (ZipArchive zip = ZipFile.Open (@"d:\zz.zip", ZipArchiveMode.Update))
  zip.CreateEntry (@"bin\X64\foo.dll").Open().Write (data, 0, data.Length);
```

También podrías hacer lo mismo completamente en memoria construyendo `ZipArchive` con un `MemoryStream`.

Operaciones de archivos y directorios

El espacio de nombres `System.IO` proporciona un conjunto de tipos para realizar operaciones "utilitarias" en archivos y directorios, como copiar y mover, crear directorios y establecer atributos y permisos de archivos. Para la mayoría de las características, puedes elegir entre dos clases, una que ofrece métodos estáticos y otra que ofrece métodos de instancia:

Clases estáticas

File y Directory

Clases de métodos de instancia (construidas con un nombre de archivo o directorio)

FileInfo y DirectoryInfo

Además, hay una clase estática llamada `Path`. Esta clase no realiza ninguna acción en archivos o directorios; en cambio, proporciona métodos de manipulación de cadenas para nombres de archivos y rutas de directorios. `Path` también ayuda con archivos temporales.

La clase File

File es una clase estática cuyos métodos aceptan un nombre de archivo. El nombre de archivo puede ser relativo al directorio actual o completamente calificado con un directorio. Aquí están sus métodos (todos públicos y estáticos):

```
bool Exists (string path);      // Returns true if the file is present

void Delete  (string path);
void Copy    (string sourceFileName, string destFileName);
void Move    (string sourceFileName, string destFileName);
void Replace (string sourceFileName, string destinationFileName,
                                     string destinationBackupFileName);

FileAttributes GetAttributes (string path);
void SetAttributes           (string path, FileAttributes fileAttributes);

void Decrypt (string path);
void Encrypt (string path);

DateTime GetCreationTime   (string path);      // UTC versions are
DateTime GetLastAccessTime (string path);      // also provided.
DateTime GetLastWriteTime  (string path);

void SetCreationTime   (string path, DateTime creationTime);
void SetLastAccessTime (string path, DateTime lastAccessTime);
void SetLastWriteTime  (string path, DateTime lastWriteTime);

FileSecurity GetAccessControl (string path);
FileSecurity GetAccessControl (string path,
                               AccessControlSections includeSections);
void SetAccessControl (string path, FileSecurity fileSecurity);
```

El método Move arroja una excepción si el archivo de destino ya existe; Replace no lo hace. Ambos métodos permiten cambiar el nombre del archivo y moverlo a otro directorio.

Delete arroja una UnauthorizedAccessException si el archivo está marcado como de solo lectura; puedes verificar esto de antemano llamando a GetAttributes. También arroja esa excepción si el sistema operativo niega los permisos de eliminación para ese archivo a tu proceso. Aquí están todos los miembros de la enumeración FileAttribute que devuelve GetAttributes:

```
Archive, Compressed, Device, Directory, Encrypted,
Hidden, IntegritySystem, Normal, NoScrubData, NotContentIndexed,
Offline, ReadOnly, ReparsePoint, SparseFile, System, Temporary
```

Los miembros en esta enumeración son combinables. Así es cómo se puede cambiar un solo atributo de archivo sin afectar los demás:

```
string filePath = "test.txt";

FileAttributes fa = File.GetAttributes (filePath);
if ((fa & FileAttributes.ReadOnly) != 0)
{
    // Use the exclusive-or operator (^) to toggle the ReadOnly flag
    fa ^= FileAttributes.ReadOnly;


     File.SetAttributes (filePath, fa);
 }

 // Now we can delete the file, for instance:
 File.Delete (filePath);
```



FileInfo ofrece una forma más sencilla de cambiar la marca de solo lectura de un archivo:

```
new FileInfo ("test.txt").IsReadOnly = false;
```

Atributos de compresión y encriptación



Esta característica es exclusiva de Windows y requiere el paquete NuGet System.Management.

Los atributos de compresión y encriptación corresponden a las casillas de verificación de compresión y encriptación en el cuadro de diálogo Propiedades de un archivo o directorio en el Explorador de Windows. Este tipo de compresión y encriptación es transparente, ya que el sistema operativo realiza todo el trabajo en segundo plano, lo que te permite leer y escribir datos sin formato.

No puedes utilizar SetAttributes para cambiar los atributos de compresión o encriptación de un archivo, ¡fallará silenciosamente si lo intentas! La solución alternativa es simple en el último caso: en su lugar, llama a los métodos Encrypt() y Decrypt() en la clase File. En el caso de la compresión, es más complicado; una solución es utilizar la API de Administración de Instrumentación de Windows (WMI) en System.Management. El siguiente método comprime un directorio y devuelve 0 si es exitoso (o un código de error de WMI si no lo es):

```
static uint CompressFolder (string folder, bool recursive)
{
  string path = "Win32_Directory.Name='" + folder + "'";
  using (ManagementObject dir = new ManagementObject (path))
  using (ManagementBaseObject p = dir.GetMethodParameters ("CompressEx"))
  {
    p ["Recursive"] = recursive;
    using (ManagementBaseObject result = dir.InvokeMethod ("CompressEx",
                                                           p, null))
      return (uint) result.Properties ["ReturnValue"].Value;
  }
}
```

Para descomprimir, reemplaza CompressEx con UncompressEx.

La encriptación transparente se basa en una clave generada a partir de la contraseña del usuario que ha iniciado sesión. El sistema es robusto ante cambios de contraseña realizados por el usuario autenticado, pero si un administrador restablece una contraseña, los datos en archivos encriptados no se pueden recuperar.



La encriptación y compresión transparentes requieren soporte especial del sistema de archivos. NTFS (utilizado comúnmente en discos duros) admite estas funciones; CDFS (en CD-ROM) y FAT (en tarjetas de medios extraíbles) no lo hacen.

Puedes determinar si un volumen admite compresión y encriptación mediante la interoperabilidad de Win32. Aquí tienes un ejemplo de cómo hacerlo:

```
using System;
using System.IO;
using System.Text;
using System.ComponentModel;
using System.Runtime.InteropServices;

class SupportsCompressionEncryption
{
  const int SupportsCompression = 0x10;
  const int SupportsEncryption = 0x20000;

  [DllImport ("Kernel32.dll", SetLastError = true)]
  extern static bool GetVolumeInformation (string vol, StringBuilder name,
    int nameSize, out uint serialNum, out uint maxNameLen, out uint flags,
    StringBuilder fileSysName, int fileSysNameSize);

  static void Main()
  {
    uint serialNum, maxNameLen, flags;
    bool ok = GetVolumeInformation (@"C:\", null, 0, out serialNum,
                                    out maxNameLen, out flags, null, 0);
    if (!ok)
      throw new Win32Exception();

    bool canCompress = (flags & SupportsCompression) != 0;
    bool canEncrypt = (flags & SupportsEncryption) != 0;
  }
}
```

Seguridad de archivos



Esta característica es exclusiva de Windows y requiere el paquete NuGet System.IO.FileSystem.AccessControl.

La clase FileSecurity te permite consultar y cambiar los permisos del sistema operativo asignados a usuarios y roles (en el espacio de nombres System.Security.AccessControl).

En el siguiente ejemplo, enumeramos los permisos existentes de un archivo y luego asignamos el permiso de escritura al grupo "Usuarios":

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

void ShowSecurity (FileSecurity sec)
{
  AuthorizationRuleCollection rules = sec.GetAccessRules (true, true,
                                                          typeof (NTAccount));
  foreach (FileSystemAccessRule r in rules.Cast<FileSystemAccessRule>()

     .OrderBy (rule => rule.IdentityReference.Value))
   {
     // e.g., MyDomain/Joe
     Console.WriteLine ($"  {r.IdentityReference.Value}");
     // Allow or Deny: e.g., FullControl
     Console.WriteLine ($"    {r.FileSystemRights}: {r.AccessControlType}");
   }
 }

 var file = "sectest.txt";
 File.WriteAllText (file, "File security test.");

 var sid = new SecurityIdentifier (WellKnownSidType.BuiltinUsersSid, null);
 string usersAccount = sid.Translate (typeof (NTAccount)).ToString();

 Console.WriteLine ($"User: {usersAccount}");

 FileSecurity sec = new FileSecurity (file,
                           AccessControlSections.Owner |
                           AccessControlSections.Group |
                           AccessControlSections.Access);

 Console.WriteLine ("AFTER CREATE:");
 ShowSecurity(sec); // BUILTIN\Users doesn't have Write permission

 sec.ModifyAccessRule (AccessControlModification.Add,
     new FileSystemAccessRule (usersAccount, FileSystemRights.Write,
                               AccessControlType.Allow),
     out bool modified);

 Console.WriteLine ("AFTER MODIFY:");
 ShowSecurity (sec); // BUILTIN\Users has Write permission
```

Damos otro ejemplo más adelante, en "Carpetas Especiales" en la página 707.

El Directorio de Clase

La clase estática Directory proporciona un conjunto de métodos análogos a los de la clase File, para verificar si un directorio existe (Exists), mover un directorio (Move), eliminar un directorio (Delete), obtener/establecer tiempos de creación o último acceso, y obtener/establecer permisos de seguridad. Además, Directory expone los siguientes métodos estáticos:

```
string GetCurrentDirectory ();
void   SetCurrentDirectory (string path);

DirectoryInfo CreateDirectory  (string path);
DirectoryInfo GetParent        (string path);
string        GetDirectoryRoot (string path);

string[] GetLogicalDrives(); // Gets mount points on Unix

// The following methods all return full paths:

string[] GetFiles             (string path);
string[] GetDirectories       (string path);
string[] GetFileSystemEntries (string path);

IEnumerable<string> EnumerateFiles             (string path);
IEnumerable<string> EnumerateDirectories       (string path);
IEnumerable<string> EnumerateFileSystemEntries (string path);
```



Los últimos tres métodos son potencialmente más eficientes que las variantes Get* porque se evalúan de forma diferida, recuperando datos del sistema de archivos mientras se enumera la secuencia. Son especialmente adecuados para consultas LINQ.

Los métodos Enumerate* y Get* también tienen sobrecargas que aceptan parámetros de searchPattern (cadena) y searchOption (enum). Si especificas SearchOption.SearchAllSubDirectories, se realizará una búsqueda recursiva en los subdirectorios. Los métodos *FileSystemEntries combinan los resultados de *Files con *Directories.

Así es cómo crear un directorio si aún no existe:

```
if (!Directory.Exists (@"d:\test"))
  Directory.CreateDirectory (@"d:\test");
```

FileInfo y DirectoryInfo

Los métodos estáticos en File y Directory son convenientes para ejecutar una sola operación de archivo o directorio. Si necesitas llamar a una serie de métodos seguidos, las clases FileInfo y DirectoryInfo proporcionan un modelo de objeto que facilita el trabajo.

FileInfo ofrece la mayoría de los métodos estáticos de File en forma de instancia, con algunas propiedades adicionales como Extension, Length, IsReadOnly y Directory, para devolver un objeto DirectoryInfo. Por ejemplo:

```
static string TestDirectory =>
  RuntimeInformation.IsOSPlatform (OSPlatform.Windows)
    ? @"C:\Temp"
    : "/tmp";

Directory.CreateDirectory (TestDirectory);

FileInfo fi = new FileInfo (Path.Combine (TestDirectory, "FileInfo.txt"));

Console.WriteLine (fi.Exists);         // false

using (TextWriter w = fi.CreateText())
  w.Write ("Some text");

Console.WriteLine (fi.Exists);         // false (still)
fi.Refresh();
Console.WriteLine (fi.Exists);         // true

Console.WriteLine (fi.Name);           // FileInfo.txt
Console.WriteLine (fi.FullName);       // c:\temp\FileInfo.txt (Windows)
                                       // /tmp/FileInfo.txt (Unix)

Console.WriteLine (fi.DirectoryName);  // c:\temp (Windows)
                                       // /tmp (Unix)
Console.WriteLine (fi.Directory.Name); // temp
Console.WriteLine (fi.Extension);      // .txt
Console.WriteLine (fi.Length);         // 9

fi.Encrypt();
fi.Attributes ^= FileAttributes.Hidden;   // (Toggle hidden flag)
fi.IsReadOnly = true;

Console.WriteLine (fi.Attributes);    // ReadOnly,Archive,Hidden,Encrypted
Console.WriteLine (fi.CreationTime);  // 3/09/2019 1:24:05 PM

fi.MoveTo (Path.Combine (TestDirectory, "FileInfoX.txt"));

DirectoryInfo di = fi.Directory;
Console.WriteLine (di.Name);             // temp or tmp
Console.WriteLine (di.FullName);         // c:\temp or /tmp
Console.WriteLine (di.Parent.FullName);  // c:\ or /
di.CreateSubdirectory ("SubFolder");
```

Aquí te mostramos cómo utilizar DirectoryInfo para enumerar archivos y subdirectorios:

```
DirectoryInfo di = new DirectoryInfo (@"e:\photos");

foreach (FileInfo fi in di.GetFiles ("*.jpg"))
  Console.WriteLine (fi.Name);

foreach (DirectoryInfo subDir in di.GetDirectories())
  Console.WriteLine (subDir.FullName);
```

Path

La clase estática Path define métodos y campos para trabajar con rutas y nombres de archivo.

Suponiendo el siguiente código de configuración:

```
string dir  = @"c:\mydir";    // or /mydir
string file = "myfile.txt";
string path = @"c:\mydir\myfile.txt";    // or /mydir/myfile.txt

Directory.SetCurrentDirectory (@"k:\demo");    // or /demo
```

Podemos demostrar los métodos y campos de Path con las siguientes expresiones:

Expression	Result (Windows, then Unix)
`Directory.GetCurrentDirectory()`	`k:\demo\` or `/demo`
`Path.IsPathRooted (file)`	`False`
`Path.IsPathRooted (path)`	`True`
`Path.GetPathRoot (path)`	`c:\` or `/`
`Path.GetDirectoryName (path)`	`c:\mydir` or `/mydir`

Expression	Result (Windows, then Unix)
`Path.GetFileName (path)`	`myfile.txt`
`Path.GetFullPath (file)`	`k:\demo\myfile.txt` or `/demo/myfile.txt`
`Path.Combine (dir, file)`	`c:\mydir\myfile.txt` or `/mydir/myfile.txt`
File extensions:	
`Path.HasExtension (file)`	`True`
`Path.GetExtension (file)`	`.txt`
`Path.GetFileNameWithoutExtension (file)`	`myfile`
`Path.ChangeExtension (file, ".log")`	`myfile.log`
Separators and characters:	
`Path.DirectorySeparatorChar`	`\` or `/`
`Path.AltDirectorySeparatorChar`	`/`
`Path.PathSeparator`	`;` or `:`
`Path.VolumeSeparatorChar`	`:` or `/`
`Path.GetInvalidPathChars()`	chars 0 to 31 and `"<>|` e or 0
`Path.GetInvalidFileNameChars()`	chars 0 to 31 and `"<>|:*?\/` or 0 and `/`
Temporary files:	
`Path.GetTempPath()`	*<local user folder>*`\Temp` or */tmp/*
`Path.GetRandomFileName()`	*`d2dwuzjf.dnp`*
`Path.GetTempFileName()`	*<local user folder>*`\Temp\tmp14B.tmp` or */tmp/`*`tmpubSUYO.tmp`

Combine es particularmente útil: te permite combinar un directorio y un nombre de archivo, o dos directorios, sin tener que verificar primero si hay un separador de ruta al final, y automáticamente utiliza el separador de ruta correcto para el sistema operativo. Proporciona sobrecargas que aceptan hasta cuatro directorios y/o nombres de archivo.

GetFullPath convierte una ruta relativa al directorio actual en una ruta absoluta. Acepta valores como ..\..\file.txt.

GetRandomFileName devuelve un nombre de archivo genuinamente único de 8.3 caracteres, sin crear realmente ningún archivo. GetTempFileName genera un nombre de archivo temporal utilizando un contador de autoincremento que se repite cada 65,000 archivos. Luego crea un archivo de cero bytes con este nombre en el directorio temporal local.



Debes eliminar el archivo generado por GetTempFileName cuando hayas terminado; de lo contrario, eventualmente lanzará una excepción (después de la llamada número 65,000 a GetTempFileName). Si esto es un problema, en su lugar puedes combinar GetTempPath con GetRandomFileName. ¡Solo ten cuidado de no llenar el disco duro del usuario!

Carpetas Especiales

Una cosa que falta en Path y Directory es una forma de localizar carpetas como Mis Documentos, Archivos de programa, Datos de la aplicación, y demás. Esto se proporciona en su lugar mediante el método GetFolderPath en la clase System.Environment:

```
string myDocPath = Environment.GetFolderPath
  (Environment.SpecialFolder.MyDocuments);
```

Environment.SpecialFolder es una enumeración cuyos valores abarcan todas las carpetas especiales en Windows, como AdminTools, ApplicationData, Fonts, History, SendTo, StartMenu, y más. Aquí se cubre todo, excepto el directorio de tiempo de ejecución de .NET, que se puede obtener de la siguiente manera:

```
System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory()
```



La mayoría de las carpetas especiales no tienen una ruta asignada en los sistemas Unix. Las siguientes tienen rutas en Ubuntu Linux 18.04 Desktop: ApplicationData, CommonApplicationData, Desktop, DesktopDirectory, LocalApplicationData, MyDocuments, MyMusic, MyPictures, MyVideos, Templates y UserProfile.

De particular valor en los sistemas Windows es ApplicationData, donde puedes almacenar configuraciones que se desplazan con un usuario a través de una red (si los perfiles de roaming están habilitados en el dominio de la red); LocalApplicationData, que es para datos no itinerantes (específicos del usuario que ha iniciado sesión); y CommonApplicationData, que es compartido por todos los usuarios de la computadora. Es considerado preferible escribir datos de la

aplicación en estas carpetas en lugar de usar el Registro de Windows. El protocolo estándar para almacenar datos en estas carpetas es crear un subdirectorio con el nombre de tu aplicación:

```
string localAppDataPath = Path.Combine (
  Environment.GetFolderPath (Environment.SpecialFolder.ApplicationData),
  "MyCoolApplication");

if (!Directory.Exists (localAppDataPath))
  Directory.CreateDirectory (localAppDataPath);
```

Hay una trampa horrible al usar CommonApplicationData: si un usuario inicia tu programa con elevación administrativa y luego tu programa crea carpetas y archivos en CommonApplicationData, ese usuario puede carecer de permisos para reemplazar esos archivos más tarde, cuando se ejecuta bajo un inicio de sesión de Windows restringido. (Un problema similar existe al cambiar entre cuentas con permisos restringidos). Puedes solucionarlo creando la carpeta deseada (con permisos asignados a todos) como parte de tu configuración.

Otro lugar para escribir archivos de configuración y registros es el directorio base de la aplicación, que puedes obtener con AppDomain.CurrentDomain.BaseDirectory. Sin embargo, no se recomienda hacerlo porque es probable que el sistema operativo niegue los permisos de escritura a tu aplicación en esta carpeta después de la instalación inicial (sin elevación administrativa).

Consulta de información de volumen

Puedes consultar las unidades de un equipo con la clase DriveInfo:

```
DriveInfo c = new DriveInfo ("C");       // Query the C: drive.
                                         // On Unix: /

long totalSize = c.TotalSize;            // Size in bytes.
long freeBytes = c.TotalFreeSpace;       // Ignores disk quotas.
long freeToMe  = c.AvailableFreeSpace;   // Takes quotas into account.

foreach (DriveInfo d in DriveInfo.GetDrives())  // All defined drives.
                                                // On Unix: mount points
{
  Console.WriteLine (d.Name);             // C:\
  Console.WriteLine (d.DriveType);        // Fixed
  Console.WriteLine (d.RootDirectory);    // C:\

  if (d.IsReady)   // If the drive is not ready, the following two
                   // properties will throw exceptions:
  {
    Console.WriteLine (d.VolumeLabel);    // The Sea Drive
    Console.WriteLine (d.DriveFormat);    // NTFS
  }
}
```

El método estático GetDrives devuelve todas las unidades mapeadas, incluyendo CD-ROMs, tarjetas de medios y conexiones de red. DriveType es un enum con los siguientes valores:

```
Unknown, NoRootDirectory, Removable, Fixed, Network, CDRom, Ram
```

Capturando eventos del sistema de archivos

La clase FileSystemWatcher te permite monitorear un directorio (y opcionalmente, subdirectorios) en busca de actividad. FileSystemWatcher tiene eventos que se activan cuando se crean, modifican, renombran o eliminan archivos o subdirectorios, así como cuando sus atributos cambian. Estos eventos se activan independientemente del usuario o proceso que realice el cambio.

Aquí tienes un ejemplo:

```
Watch (GetTestDirectory(), "*.txt", true);

void Watch (string path, string filter, bool includeSubDirs)
{

  using (var watcher = new FileSystemWatcher (path, filter))
  {
    watcher.Created += FileCreatedChangedDeleted;
    watcher.Changed += FileCreatedChangedDeleted;
    watcher.Deleted += FileCreatedChangedDeleted;
    watcher.Renamed += FileRenamed;
    watcher.Error   += FileError;

    watcher.IncludeSubdirectories = includeSubDirs;
    watcher.EnableRaisingEvents = true;

    Console.WriteLine ("Listening for events - press <enter> to end");
    Console.ReadLine();
  }
  // Disposing the FileSystemWatcher stops further events from firing.
}

void FileCreatedChangedDeleted (object o, FileSystemEventArgs e)
  => Console.WriteLine ("File {0} has been {1}", e.FullPath, e.ChangeType);

void FileRenamed (object o, RenamedEventArgs e)
  => Console.WriteLine ("Renamed: {0}->{1}", e.OldFullPath, e.FullPath);

void FileError (object o, ErrorEventArgs e)
  => Console.WriteLine ("Error: " + e.GetException().Message);

string GetTestDirectory() =>
  RuntimeInformation.IsOSPlatform (OSPlatform.Windows)
    ? @"C:\Temp"
    : "/tmp";
```



Debido a que FileSystemWatcher genera eventos en un hilo separado, debes manejar las excepciones en el código de manejo de eventos para evitar que un error haga que la aplicación se cierre. Para obtener más información, consulta "Manejo de Excepciones" en la página 618.

El evento Error no te informa sobre errores del sistema de archivos; en su lugar, indica que el búfer de eventos de FileSystemWatcher se desbordó debido a una gran cantidad de eventos

Changed, Created, Deleted o Renamed. Puedes cambiar el tamaño del búfer a través de la propiedad InternalBufferSize.

El parámetro IncludeSubdirectories se aplica de forma recursiva. Por lo tanto, si creas un FileSystemWatcher en C:\ con IncludeSubdirectories establecido en true, sus eventos se activarán cuando un archivo o directorio cambie en cualquier lugar del disco duro.



Un problema al usar FileSystemWatcher es abrir y leer archivos recién creados o actualizados antes de que el archivo se haya poblado o actualizado por completo. Si estás trabajando en conjunto con otro software que crea archivos, es posible que debas considerar alguna estrategia para mitigar esto, como crear archivos con una extensión no monitoreada y luego cambiarles el nombre una vez que estén completamente escritos.

Seguridad del sistema operativo

Todas las aplicaciones están sujetas a restricciones del sistema operativo basadas en los privilegios de inicio de sesión del usuario. Estas restricciones afectan la operación de archivos, así como otras capacidades, como el acceso al Registro de Windows.

En Windows y Unix, existen dos tipos de cuentas:

- Una cuenta de administrador/superusuario que no impone restricciones en el acceso a la computadora local.
- Una cuenta con permisos limitados que restringe las funciones administrativas y la visibilidad de los datos de otros usuarios.

En Windows, una característica llamada Control de cuentas de usuario (UAC, por sus siglas en inglés) significa que los administradores reciben dos tokens, o "sombreros", al iniciar sesión: un sombrero de administrador y un sombrero de usuario normal. Por defecto, los programas se ejecutan usando el sombrero de usuario normal, con permisos restringidos, a menos que el programa solicite elevación administrativa. El usuario debe aprobar la solicitud en el cuadro de diálogo que se presenta.

En Unix, los usuarios suelen iniciar sesión con cuentas restringidas. Esto también se aplica a los administradores para reducir la probabilidad de dañar accidentalmente el sistema. Cuando un usuario necesita ejecutar un comando que requiere permisos elevados, preceden el comando con "sudo" (abreviatura de superusuario).

Por defecto, tu aplicación se ejecutará con privilegios de usuario restringidos. Esto significa que debes hacer una de las siguientes opciones:

- Escribir tu aplicación de manera que pueda ejecutarse sin privilegios administrativos.
- Requerir elevación administrativa en el manifiesto de la aplicación (solo en Windows) o detectar la falta de privilegios necesarios y alertar al usuario para reiniciar la aplicación como administrador/superusuario.

La primera opción es más segura y conveniente para el usuario. Diseñar tu programa para funcionar sin privilegios administrativos es fácil en la mayoría de los casos.

Puedes determinar si estás ejecutando la aplicación bajo una cuenta administrativa de la siguiente manera:

```
[DllImport("libc")]
public static extern uint getuid();

static bool IsRunningAsAdmin()
{
  if (RuntimeInformation.IsOSPlatform (OSPlatform.Windows))
  {
    using var identity = WindowsIdentity.GetCurrent();
    var principal = new WindowsPrincipal (identity);
    return principal.IsInRole (WindowsBuiltInRole.Administrator);
  }

  return getuid() == 0;
}
```

Con el UAC habilitado en Windows, esto devuelve true solo si el proceso actual tiene elevación administrativa. En Linux, devuelve true solo si el proceso actual se está ejecutando como superusuario (por ejemplo, "sudo myapp").

Ejecutándose en una cuenta de usuario estándar

Estas son las principales acciones que no se pueden realizar en una cuenta de usuario estándar:

- Escribir en los siguientes directorios:
  - La carpeta del sistema operativo (normalmente \Windows o /bin, /sbin, ...) y sus subdirectorios.
  - La carpeta de archivos de programas (\Program Files o /usr/bin, /opt) y sus subdirectorios.
  - La raíz de la unidad del sistema operativo (por ejemplo, C:\ o /).
- Escribir en la rama HKEY_LOCAL_MACHINE del Registro (Windows).
- Leer datos de monitoreo de rendimiento (WMI) (Windows).

Además, como usuario ordinario de Windows (incluso como administrador), es posible que se te deniegue el acceso a archivos o recursos que pertenecen a otros usuarios. Windows utiliza un sistema de Listas de Control de Acceso (ACL) para proteger dichos recursos; puedes consultar y afirmar tus propios derechos en las ACL mediante los tipos en System.Security.AccessControl. Las ACL también se pueden aplicar a los identificadores de espera entre procesos, que se describen en el Capítulo 21.

Si se te deniega el acceso a algo como resultado de la seguridad del sistema operativo, el CLR detectará el fallo y generará una UnauthorizedAccessException (en lugar de fallar silenciosamente).

En la mayoría de los casos, puedes manejar las restricciones de la cuenta de usuario estándar de la siguiente manera:

- Escribir archivos en sus ubicaciones recomendadas.

- Evitar utilizar el Registro para información que se pueda almacenar en archivos (excepto el contenedor HKEY_CURRENT_USER, al que tendrás acceso de lectura/escritura solo en Windows).
- Registrar componentes ActiveX o COM durante la instalación (solo en Windows).

La ubicación recomendada para los documentos del usuario es SpecialFolder.MyDocuments:

```
string docsFolder = Environment.GetFolderPath
                    (Environment.SpecialFolder.MyDocuments);

string path = Path.Combine (docsFolder, "test.txt");
```

La ubicación recomendada para los archivos de configuración que un usuario pueda necesitar modificar fuera de tu aplicación es SpecialFolder.ApplicationData (solo para el usuario actual) o SpecialFolder.CommonApplicationData (para todos los usuarios). Por lo general, creas subdirectorios dentro de estas carpetas basándote en el nombre de tu organización y producto.

Elevación administrativa y virtualización

Con un manifiesto de aplicación, puedes solicitar que Windows solicite al usuario una elevación administrativa cada vez que se ejecute tu programa (Linux ignora esta solicitud). Aquí tienes un ejemplo de un manifiesto de aplicación:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="requireAdministrator" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

(Describimos los manifiestos de aplicaciones con más detalle en el Capítulo 17.)

Si reemplazas `requireAdministrator` con `asInvoker`, indicas a Windows que no se requiere elevación administrativa. El efecto es casi el mismo que no tener un manifiesto de aplicación en absoluto, excepto que la virtualización se desactiva. La virtualización es una medida temporal introducida con Windows Vista para ayudar a que las aplicaciones antiguas se ejecuten correctamente sin privilegios administrativos. La ausencia de un manifiesto de aplicación con el elemento `requestedExecutionLevel` activa esta característica de compatibilidad con versiones anteriores.

La virtualización entra en juego cuando una aplicación escribe en el directorio Program Files o Windows, o en el área HKEY_LOCAL_MACHINE del Registro. En lugar de generar una excepción, los cambios se redirigen a una ubicación separada en el disco duro donde no pueden afectar los datos originales. Esto evita que la aplicación interfiera con el sistema operativo u otras aplicaciones bien comportadas.

Archivos mapeados en memoria

Los archivos mapeados en memoria proporcionan dos características clave:

- Acceso aleatorio eficiente a los datos del archivo.
- La capacidad de compartir memoria entre diferentes procesos en la misma computadora.

Los tipos para los archivos mapeados en memoria se encuentran en el espacio de nombres System.IO.MemoryMappedFiles. Internamente, funcionan envolviendo la API del sistema operativo para archivos mapeados en memoria.

Los archivos mapeados en memoria y la E/S de archivos aleatoria

Aunque un FileStream ordinario permite la E/S de archivos aleatoria (configurando la propiedad Position del flujo), está optimizado para la E/S secuencial. Como regla general aproximada:

- Los FileStreams son aproximadamente 10 veces más rápidos que los archivos mapeados en memoria para la E/S secuencial.
- Los archivos mapeados en memoria son aproximadamente 10 veces más rápidos que los FileStreams para la E/S aleatoria.

Cambiar la posición de un FileStream puede costar varios microsegundos, lo cual se acumula si se realiza dentro de un bucle. Además, un FileStream no es adecuado para acceso simultáneo de subprocesos, ya que su posición cambia a medida que se lee o se escribe en él.

Para crear un archivo mapeado en memoria:

1. Obtén un FileStream como lo harías normalmente.

2. Instancia un MemoryMappedFile, pasando el FileStream.

3. Llama a CreateViewAccessor en el objeto de archivo mapeado en memoria.

El último paso te proporciona un objeto MemoryMappedViewAccessor que ofrece métodos para leer y escribir de forma aleatoria tipos simples, estructuras y matrices (más detalles en "Trabajando con Accessors de vista" en la página 715).

El siguiente ejemplo crea un archivo de un millón de bytes y luego utiliza la API de archivos mapeados en memoria para leer y luego escribir un byte en la posición 500,000:

```
File.WriteAllBytes ("long.bin", new byte [1000000]);

using MemoryMappedFile mmf = MemoryMappedFile.CreateFromFile ("long.bin");
using MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor();

accessor.Write (500000, (byte) 77);
Console.WriteLine (accessor.ReadByte (500000));   // 77
```

También puedes especificar un nombre de mapeo y capacidad al llamar a CreateFromFile. Especificar un nombre de mapeo no nulo permite que el bloque de memoria se comparta con otros procesos (ver la siguiente sección); especificar una capacidad amplía automáticamente el archivo a ese valor. El siguiente ejemplo crea un archivo de 1,000 bytes:

```
File.WriteAllBytes ("short.bin", new byte [1]);
using (var mmf = MemoryMappedFile.CreateFromFile
                 ("short.bin", FileMode.Create, null, 1000))
  ...
```

Archivos mapeados en memoria y memoria compartida (Windows)

En Windows, también puedes utilizar archivos mapeados en memoria como medio para compartir memoria entre procesos en la misma computadora. Un proceso crea un bloque de memoria compartida llamando a MemoryMappedFile.CreateNew, y luego otros procesos se suscriben a ese mismo bloque de memoria llamando a MemoryMappedFile.OpenExisting con el mismo nombre. Aunque aún se le llama "archivo" mapeado en memoria, reside completamente en la memoria y no tiene presencia en disco.

El siguiente código crea un archivo mapeado en memoria compartida de 500 bytes y escribe el entero 12345 en la posición 0:

```
using (MemoryMappedFile mmFile = MemoryMappedFile.CreateNew ("Demo", 500))
using (MemoryMappedViewAccessor accessor = mmFile.CreateViewAccessor())
{
  accessor.Write (0, 12345);
  Console.ReadLine();   // Keep shared memory alive until user hits Enter.
}
```

El siguiente código abre ese archivo mapeado en memoria y lee ese entero:

```
// This can run in a separate executable:
using (MemoryMappedFile mmFile = MemoryMappedFile.OpenExisting ("Demo"))
using (MemoryMappedViewAccessor accessor = mmFile.CreateViewAccessor())
  Console.WriteLine (accessor.ReadInt32 (0));   // 12345
```

Memoria compartida interprocesos multiplataforma

Tanto Windows como Unix permiten que múltiples procesos mapeen en memoria el mismo archivo. Debes tener cuidado para asegurar la configuración adecuada de uso compartido de archivos:

```
static void Writer()
{
  var file = Path.Combine (TestDirectory, "interprocess.bin");
  File.WriteAllBytes (file, new byte [100]);

  using FileStream fs =
    new FileStream (file, FileMode.Open, FileAccess.ReadWrite,
                    FileShare.ReadWrite);

  using MemoryMappedFile mmf = MemoryMappedFile
    .CreateFromFile (fs, null, fs.Length, MemoryMappedFileAccess.ReadWrite,
                     HandleInheritability.None, true);
  using MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor();

  accessor.Write (0, 12345);

  Console.ReadLine();   // Keep shared memory alive until user hits Enter.

  File.Delete (file);
}

static void Reader()
{
  // This can run in a separate executable:
  var file = Path.Combine (TestDirectory, "interprocess.bin");
  using FileStream fs =
    new FileStream (file, FileMode.Open, FileAccess.ReadWrite,
                    FileShare.ReadWrite);
  using MemoryMappedFile mmf = MemoryMappedFile

     .CreateFromFile (fs, null, fs.Length, MemoryMappedFileAccess.ReadWrite,
                      HandleInheritability.None, true);
   using MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor();

   Console.WriteLine (accessor.ReadInt32 (0));   // 12345
}

static string TestDirectory =>
  RuntimeInformation.IsOSPlatform (OSPlatform.Windows)
    ?  @"C:\Test"
    : "/tmp";
```

Trabajar con accesores de vista

Llamar a CreateViewAccessor en un MemoryMappedFile te proporciona un accesor de vista que te permite leer/escribir valores en posiciones aleatorias.

Los métodos Read*/Write* aceptan tipos numéricos, bool y char, así como matrices y estructuras que contienen elementos o campos de tipo valor. Los tipos de referencia, así como las matrices y estructuras que contienen tipos de referencia, están prohibidos porque no se pueden mapear en memoria no administrada. Entonces, si quieres escribir una cadena, debes codificarla en una matriz de bytes:

```
byte[] data = Encoding.UTF8.GetBytes ("This is a test");
accessor.Write (0, data.Length);
accessor.WriteArray (4, data, 0, data.Length);
```

Observa que primero escribimos la longitud. Esto significa que sabemos cuántos bytes leer más adelante:

```
byte[] data = new byte [accessor.ReadInt32 (0)];
accessor.ReadArray (4, data, 0, data.Length);
Console.WriteLine (Encoding.UTF8.GetString (data));   // This is a test
```

Aquí tienes un ejemplo de lectura/escritura de una estructura:

```
struct Data { public int X, Y; }
...
var data = new Data { X = 123, Y = 456 };
accessor.Write (0, ref data);
accessor.Read (0, out data);
Console.WriteLine (data.X + " " + data.Y);   // 123 456
```

Los métodos Read y Write son sorprendentemente lentos. Puedes obtener un rendimiento mucho mejor accediendo directamente a la memoria no administrada subyacente mediante un puntero. Siguiendo con el ejemplo anterior:

```
unsafe
{
  byte* pointer = null;
  try
  {
    accessor.SafeMemoryMappedViewHandle.AcquirePointer (ref pointer);
    int* intPointer = (int*) pointer;
    Console.WriteLine (*intPointer);               // 123


  }
  finally
  {
    if (pointer != null)
      accessor.SafeMemoryMappedViewHandle.ReleasePointer();
  }
}
```

Tu proyecto debe estar configurado para permitir código inseguro. Puedes hacerlo editando tu archivo .csproj de la siguiente manera:

```
<PropertyGroup>
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

La ventaja de rendimiento de los punteros es aún más notable cuando se trabaja con estructuras grandes, ya que te permiten trabajar directamente con los datos sin procesar en lugar de utilizar Read/Write para copiar datos entre memoria administrada y no administrada. Exploraremos esto en más detalle en el Capítulo 24.

16 Redes (Networking)

.NET ofrece una variedad de clases en los espacios de nombres System.Net.* para comunicarse a través de protocolos de red estándar, como HTTP y TCP/IP. A continuación se muestra un resumen de los componentes clave:

- HttpClient para consumir APIs web HTTP y servicios RESTful.
- HttpListener para crear un servidor HTTP.
- SmtpClient para construir y enviar mensajes de correo electrónico mediante SMTP.
- Dns para convertir entre nombres de dominio y direcciones IP.
- TcpClient, UdpClient, TcpListener y Socket para acceder directamente a las capas de transporte y red.

Los tipos .NET en este capítulo se encuentran en los espacios de nombres System.Net.* y System.IO.



.NET también proporciona soporte del lado del cliente para FTP, pero solo a través de clases que se han marcado como obsoletas a partir de .NET 6. Si necesitas utilizar FTP, la mejor opción es buscar una biblioteca NuGet como FluentFTP.

Arquitectura de red (Network Architecture)

La figura 16-1 ilustra los tipos de networking de .NET y las capas de comunicación en las que residen. La mayoría de los tipos se encuentran en la capa de transporte o en la capa de aplicación. La capa de transporte define protocolos básicos para enviar y recibir bytes (TCP y UDP), mientras que la capa de aplicación define protocolos de nivel superior diseñados para aplicaciones específicas, como recuperar páginas web (HTTP), enviar correos electrónicos (SMTP) y convertir entre nombres de dominio y direcciones IP (DNS).

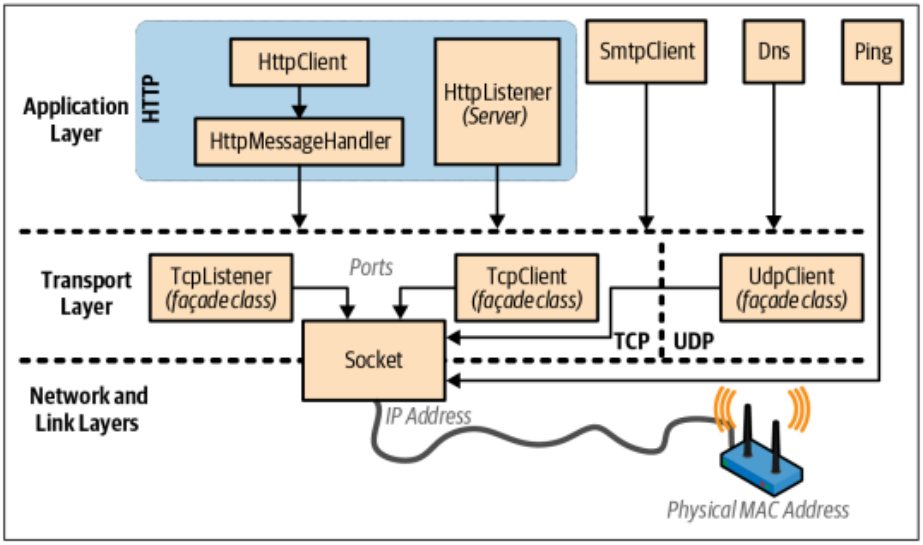


Figure 16-1. Network architecture

xSuele ser más conveniente programar en la capa de aplicación; sin embargo, hay algunas razones por las cuales podrías querer trabajar directamente en la capa de transporte. Una de ellas es si necesitas un protocolo de aplicación que no está provisto en .NET, como POP3 para recuperar correos electrónicos. Otra razón es si deseas inventar un protocolo personalizado para una aplicación especial, como un cliente peer-to-peer.

De los protocolos de aplicación, HTTP es especial en su aplicabilidad a la comunicación de propósito general. Su modo básico de operación, "dame la página web con esta URL", se adapta fácilmente a "dame el resultado de llamar a este punto final con estos argumentos". (Además del verbo "get", también existen "put", "post" y "delete", lo que permite servicios basados en REST).

HTTP también cuenta con un conjunto completo de características que son útiles en aplicaciones empresariales de múltiples niveles y en arquitecturas orientadas a servicios, como protocolos de autenticación y cifrado, fragmentación de mensajes, encabezados y cookies extensibles, y la capacidad de que muchas aplicaciones servidoras compartan un solo puerto y dirección IP. Por estas razones, HTTP está bien soportado en .NET, tanto directamente, como se describe en este capítulo, como a un nivel más alto, a través de tecnologías como Web API y ASP.NET Core.

Como se ha dejado claro en la discusión anterior, las redes son un campo que está inundado de acrónimos. Enumeramos los más comunes en la Tabla 16-1.

Table 16-1. Network acronyms

Acronym	Expansion	Notes
DNS	Domain Name Service	Converts between domain names (e.g., *ebay.com*) and IP addresses (e.g., 199.54.213.2)
FTP	File Transfer Protocol	Internet-based protocol for sending and receiving files
HTTP	Hypertext Transfer Protocol	Retrieves web pages and runs web services
IIS	Internet Information Services	Microsoft's web server software
IP	Internet Protocol	Network-layer protocol below TCP and UDP
LAN	Local Area Network	Most LANs use internet-based protocols such as TCP/IP
POP	Post Office Protocol	Retrieves internet mail
REST	REpresentational State Transfer	A popular web service architecture that uses machine-followable links in responses and that can operate over basic HTTP
SMTP	Simple Mail Transfer Protocol	Sends internet mail
TCP	Transmission and Control Protocol	Transport-layer internet protocol on top of which most higher-layer services are built
UDP	Universal Datagram Protocol	Transport-layer internet protocol used for low-overhead services such as VoIP
UNC	Universal Naming Convention	*\\computer\sharename\filename*
URI	Uniform Resource Identifier	Ubiquitous resource naming system (e.g., *http://www.amazon.com* or *mailto:joe@bloggs.org*)
URL	Uniform Resource Locator	Technical meaning (fading from use): subset of URI; popular meaning: synonym of URI

Direcciones y Puertos (Addresses and Ports)

Para que la comunicación funcione, una computadora o dispositivo requiere una dirección. Internet utiliza dos sistemas de direccionamiento:

IPv4

Actualmente es el sistema de direccionamiento dominante; las direcciones IPv4 tienen 32 bits de ancho. Cuando se formatean como cadena de texto, las direcciones IPv4 se escriben como cuatro números decimales separados por puntos (por ejemplo, 101.102.103.104). Una dirección puede ser única en el mundo o única dentro de una subred en particular (como en una red corporativa).

IPv6

Es el sistema de direccionamiento más nuevo y utiliza direcciones de 128 bits. Las direcciones se formatean como cadenas de texto en hexadecimal con dos puntos de separación (por ejemplo, [3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]). En .NET, se requiere que se agreguen corchetes alrededor de la dirección.

La clase IPAddress en el espacio de nombres System.Net representa una dirección en cualquiera de los dos protocolos. Tiene un constructor que acepta un arreglo de bytes y un método estático Parse que acepta una cadena de texto correctamente formateada:

```
IPAddress a1 = new IPAddress (new byte[] { 101, 102, 103, 104 });
IPAddress a2 = IPAddress.Parse ("101.102.103.104");
Console.WriteLine (a1.Equals (a2));                    // True


Console.WriteLine (a1.AddressFamily);                  // InterNetwork

IPAddress a3 = IPAddress.Parse
  ("[3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]");
Console.WriteLine (a3.AddressFamily);   // InterNetworkV6
```

Los protocolos TCP y UDP dividen cada dirección IP en 65,535 puertos, lo que permite que una computadora en una sola dirección ejecute múltiples aplicaciones, cada una en su propio puerto. Muchas aplicaciones tienen asignaciones de puertos predeterminadas estándar; por ejemplo, HTTP utiliza el puerto 80 y SMTP utiliza el puerto 25.



Los puertos TCP y UDP del 49152 al 65535 no están asignados oficialmente, por lo que son buenos para pruebas e implementaciones a pequeña escala.

Una combinación de dirección IP y puerto se representa en .NET mediante la clase IPEndPoint:

```
IPAddress a = IPAddress.Parse ("101.102.103.104");
IPEndPoint ep = new IPEndPoint (a, 222);          // Port 222
Console.WriteLine (ep.ToString());                // 101.102.103.104:222
```



Los firewalls bloquean los puertos. En muchos entornos corporativos, solo están abiertos algunos puertos, típicamente el puerto 80 (para HTTP no encriptado) y el puerto 443 (para HTTPS seguro).

URIs

Un URI es una cadena con un formato especial que describe un recurso en Internet o en una LAN, como una página web, un archivo o una dirección de correo electrónico. Ejemplos incluyen http://www.ietf.org, ftp://myisp/doc.txt y mailto:joe@bloggs.com. El formato exacto está definido por el Grupo de Trabajo de Ingeniería de Internet (IETF).

Un URI se puede descomponer en una serie de elementos, típicamente esquema, autoridad y ruta. La clase Uri en el espacio de nombres System realiza esta división, exponiendo una propiedad para cada elemento, como se ilustra en la Figura 16-2.

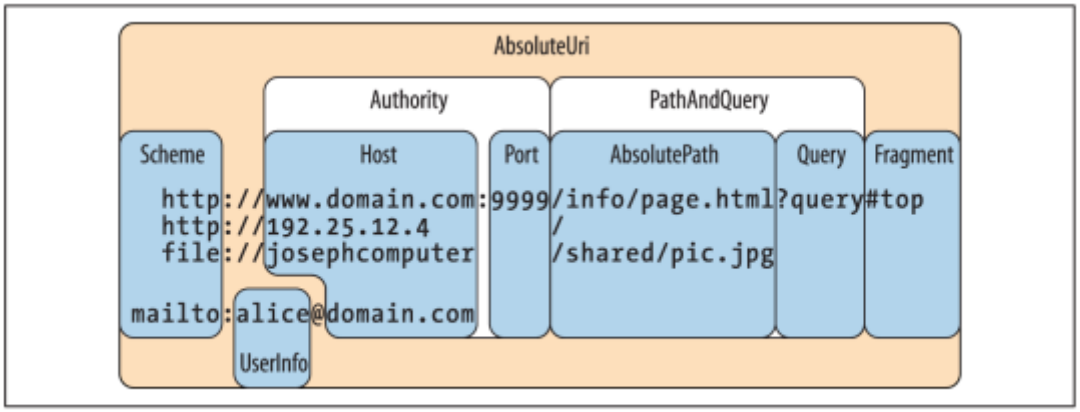


Figure 16-2. URI properties



La clase Uri es útil cuando necesitas validar el formato de una cadena URI o dividir una URI en sus partes componentes. De lo contrario, puedes tratar una URI simplemente como una cadena, ya que la mayoría de los métodos de red están sobrecargados para aceptar tanto un objeto Uri como una cadena.

Puedes construir un objeto Uri pasando cualquiera de las siguientes cadenas a su constructor:

- Una cadena URI, como http://www.ebay.com o file://janespc/sharedpics/dolphin.jpg
- Una ruta absoluta a un archivo en tu disco duro, como c:\myfiles\data.xlsx o, en Unix, /tmp/myfiles/data.xlsx
- Una ruta UNC a un archivo en la LAN, como \\janespc\sharedpics\dolphin.jpg

Las rutas de archivo y UNC se convierten automáticamente a URIs: se agrega el protocolo "file:" y las barras invertidas se convierten en barras inclinadas. Los constructores de Uri también realizan algunas limpiezas básicas en tu cadena antes de crear el Uri, incluyendo la conversión del esquema y el nombre de host a minúsculas, y la eliminación de números de puerto

predeterminados y en blanco. Si proporcionas una cadena URI sin el esquema, como www.test.com, se lanzará una UriFormatException.

Uri tiene una propiedad IsLoopback, que indica si la Uri hace referencia al host local (dirección IP 127.0.0.1), y una propiedad IsFile, que indica si la Uri hace referencia a una ruta local o UNC (IsUnc informa false para un recurso compartido de Samba montado en un sistema de archivos Linux). Si IsFile devuelve true, la propiedad LocalPath devuelve una versión de AbsolutePath que es compatible con el sistema operativo local (con barras inclinadas o barras invertidas según corresponda al sistema operativo), en la cual se puede llamar a File.Open.

Las instancias de Uri tienen propiedades de solo lectura. Para modificar un Uri existente, puedes instanciar un objeto UriBuilder, el cual tiene propiedades modificables y se puede convertir de vuelta a través de su propiedad Uri.

Uri también proporciona métodos para comparar y restar rutas:

```
Uri info = new Uri ("http://www.domain.com:80/info/");
Uri page = new Uri ("http://www.domain.com/info/page.html");

Console.WriteLine (info.Host);      // www.domain.com
Console.WriteLine (info.Port);      // 80
Console.WriteLine (page.Port);      // 80  (Uri knows the default HTTP port)

Console.WriteLine (info.IsBaseOf (page));         // True
Uri relative = info.MakeRelativeUri (page);
Console.WriteLine (relative.IsAbsoluteUri);       // False
Console.WriteLine (relative.ToString());          // page.html
```

Un Uri relativo, como "page.html" en este ejemplo, lanzará una excepción si llamas a cualquier propiedad o método que no sea IsAbsoluteUri y ToString(). Puedes instanciar directamente un Uri relativo de la siguiente manera:

```
Uri u = new Uri ("page.html", UriKind.Relative);
```

 Una barra inclinada al final es significativa en una URI y marca una diferencia en cómo un servidor procesa una solicitud si hay un componente de ruta presente.

En un servidor web tradicional, por ejemplo, dado el URI http://www.albahari.com/nutshell/, puedes esperar que un servidor web HTTP busque en el subdirectorio "nutshell" en la carpeta web del sitio y devuelva el documento predeterminado (generalmente index.html). Sin la barra inclinada al final, el servidor web buscará en cambio un archivo llamado "nutshell" (sin extensión) directamente en la carpeta raíz del sitio, lo cual generalmente no es lo que deseas. Si no existe dicho archivo, la mayoría de los servidores web asumirán que el usuario cometió un error y devolverán un error de redirección permanente 301, sugiriendo al cliente que vuelva a intentarlo con la barra inclinada al final. Un cliente HTTP de .NET, por defecto, responderá de manera transparente a un 301 de la misma manera que un navegador web, volviendo a intentarlo con el URI sugerido. Esto significa que si omites una barra inclinada al final cuando debería haber estado incluida, tu solicitud seguirá funcionando, pero sufrirá una vuelta de más innecesaria.

La clase Uri también proporciona métodos de ayuda estáticos, como EscapeUriString(), que convierte una cadena en una URL válida al convertir todos los caracteres con un valor ASCII mayor que 127 en su representación hexadecimal. Los métodos CheckHostName() y CheckSchemeName() aceptan una cadena y verifican si es sintácticamente válida para la propiedad dada (aunque no intentan determinar si existe un host o una URI).

HttpClient

La clase HttpClient expone una API moderna para operaciones de cliente HTTP, reemplazando los antiguos tipos WebClient y WebRequest/WebResponse (que han sido marcados como obsoletos).

HttpClient fue escrita en respuesta al crecimiento de las API web basadas en HTTP y los servicios REST, y proporciona una buena experiencia al trabajar con protocolos más elaborados que simplemente obtener una página web. En particular:

- Una instancia de HttpClient puede manejar solicitudes concurrentes y funciona bien con características como encabezados personalizados, cookies y esquemas de autenticación.
- HttpClient te permite escribir e incorporar controladores de mensajes personalizados. Esto permite realizar simulaciones en pruebas unitarias y crear tuberías personalizadas (para registro, compresión, encriptación, etc.).
- HttpClient tiene un sistema de tipos rico y extensible para encabezados y contenido.



HttpClient no admite informes de progreso. Para una solución, consulta "HttpClient with Progress.linq" en http://www.albahari.com/nutshell/code.aspx o a través de la galería de muestras interactivas de LINQPad.

La forma más sencilla de usar HttpClient es instanciarlo y luego llamar a uno de sus métodos Get*, pasando un URI:

```
string html = await new HttpClient().GetStringAsync ("http://linqpad.net");
```

(También existen GetByteArrayAsync y GetStreamAsync). Todos los métodos de E/S de HttpClient están diseñados para ser asíncronos.

A diferencia de sus predecesores WebRequest/WebResponse, para obtener el mejor rendimiento con HttpClient, debes reutilizar la misma instancia (de lo contrario, cosas como la resolución de DNS pueden repetirse innecesariamente y los sockets se mantendrán abiertos más tiempo del necesario). HttpClient permite operaciones concurrentes, por lo que el siguiente código es válido y descarga dos páginas web al mismo tiempo:

```
var client = new HttpClient();
var task1 = client.GetStringAsync ("http://www.linqpad.net");
var task2 = client.GetStringAsync ("http://www.albahari.com");
Console.WriteLine (await task1);
Console.WriteLine (await task2);
```

HttpClient tiene una propiedad Timeout y una propiedad BaseAddress, que agrega un URI a cada solicitud. HttpClient es algo así como una envoltura delgada: la mayoría de las otras propiedades

que esperarías encontrar aquí están definidas en otra clase llamada HttpClientHandler. Para acceder a esta clase, la instancias y luego la pasas al constructor de HttpClient:

```
var handler = new HttpClientHandler { UseProxy = false };
var client = new HttpClient (handler);
...
```

En este ejemplo, le indicamos al controlador que deshabilite el soporte de proxy, lo que a veces puede mejorar el rendimiento al evitar el costo de la detección automática de proxy. También hay propiedades para controlar cookies, redirección automática, autenticación, etc. (describiremos estas en las secciones siguientes).

GetAsync y Mensajes de Respuesta

Los métodos GetStringAsync, GetByteArrayAsync y GetStreamAsync son accesos directos convenientes para llamar al método más general GetAsync, que devuelve un mensaje de respuesta:

```
var client = new HttpClient();
// The GetAsync method also accepts a CancellationToken.
HttpResponseMessage response = await client.GetAsync ("http://...");

response.EnsureSuccessStatusCode();
string html = await response.Content.ReadAsStringAsync();
```

HttpResponseMessage expone propiedades para acceder a los encabezados (ver "Headers" en la página 729) y al código de estado HTTP. Un código de estado no exitoso, como 404 (no encontrado), no genera una excepción a menos que llames explícitamente a EnsureSuccessStatusCode. Sin embargo, los errores de comunicación o DNS sí generan excepciones.

HttpContent tiene un método CopyToAsync para escribir en otro flujo, lo cual es útil para escribir la salida en un archivo:

```
using (var fileStream = File.Create ("linqpad.html"))
  await response.Content.CopyToAsync (fileStream);
```

GetAsync es uno de los cuatro métodos que corresponden a los cuatro verbos HTTP (los otros son PostAsync, PutAsync y DeleteAsync). Demostraremos PostAsync más adelante en "Carga de datos de formulario" en la página 730.

SendAsync y mensajes de solicitud

GetAsync, PostAsync, PutAsync y DeleteAsync son accesos directos para llamar a SendAsync, el método de bajo nivel en el que todo se alimenta. Para usarlo, primero construyes un objeto HttpRequestMessage:

```
var client = new HttpClient();
var request = new HttpRequestMessage (HttpMethod.Get, "http://...");
HttpResponseMessage response = await client.SendAsync (request);
response.EnsureSuccessStatusCode();
...
```

La instancia de un objeto HttpRequestMessage te permite personalizar propiedades de la solicitud, como los encabezados (ver "Headers" en la página 729) y el contenido mismo, lo que te permite cargar datos.

Carga de datos y HttpContent

Después de instanciar un objeto HttpRequestMessage, puedes cargar contenido asignando su propiedad Content. El tipo para esta propiedad es una clase abstracta llamada HttpContent. .NET incluye las siguientes subclases concretas para diferentes tipos de contenido (también puedes escribir la tuya propia):

- ByteArrayContent
- StringContent
- FormUrlEncodedContent (ver "Carga de datos de formulario" en la página 730)
- StreamContent

Por ejemplo:

```
var client = new HttpClient (new HttpClientHandler { UseProxy = false });
var request = new HttpRequestMessage (
  HttpMethod.Post, "http://www.albahari.com/EchoPost.aspx");
request.Content = new StringContent ("This is a test");
HttpResponseMessage response = await client.SendAsync (request);
response.EnsureSuccessStatusCode();
Console.WriteLine (await response.Content.ReadAsStringAsync());
```

HttpMessageHandler

Dijimos anteriormente que la mayoría de las propiedades para personalizar las solicitudes están definidas no en HttpClient, sino en HttpClientHandler. Este último es en realidad una subclase de la clase abstracta HttpMessageHandler, definida de la siguiente manera:

```
public abstract class HttpMessageHandler : IDisposable
{
  protected internal abstract Task<HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken);

  public void Dispose();
  protected virtual void Dispose (bool disposing);
}
```

El método SendAsync se llama desde el método SendAsync de HttpClient.

HttpMessageHandler es lo suficientemente simple como para ser subclasificada fácilmente y ofrece un punto de extensibilidad en HttpClient.

Pruebas unitarias y simulación

Podemos subclasificar HttpMessageHandler para crear un controlador simulado que ayude con las pruebas unitarias:

```
class MockHandler : HttpMessageHandler
{
  Func <HttpRequestMessage, HttpResponseMessage> _responseGenerator;

  public MockHandler
    (Func <HttpRequestMessage, HttpResponseMessage> responseGenerator)
  {
    _responseGenerator = responseGenerator;
  }

  protected override Task <HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken)
  {
    cancellationToken.ThrowIfCancellationRequested();
    var response = _responseGenerator (request);
    response.RequestMessage = request;
    return Task.FromResult (response);
  }
}
```

Su constructor acepta una función que le indica al simulador cómo generar una respuesta a partir de una solicitud. Este enfoque es el más versátil, ya que el mismo controlador puede probar múltiples solicitudes.

SendAsync es síncrono gracias a Task.FromResult. Podríamos haber mantenido la asincronía haciendo que nuestro generador de respuestas devuelva una Task<HttpResponseMessage>, pero esto no tiene sentido dado que podemos esperar que una función de simulación se ejecute rápidamente. Así es cómo se usa nuestro controlador de simulación:

```
var mocker = new MockHandler (request =>
  new HttpResponseMessage (HttpStatusCode.OK)
  {
    Content = new StringContent ("You asked for " + request.RequestUri)
  });

var client = new HttpClient (mocker);
var response = await client.GetAsync ("http://www.linqpad.net");
string result = await response.Content.ReadAsStringAsync();
Assert.AreEqual ("You asked for http://www.linqpad.net/", result);
```

Encadenando controladores con DelegatingHandler

Puedes crear un controlador de mensajes que llame a otro (resultando en una cadena de controladores) mediante la subclase DelegatingHandler. Puedes usar esto para implementar protocolos de autenticación personalizados, compresión y encriptación. A continuación se muestra un ejemplo de un controlador simple de registro:

```
class LoggingHandler : DelegatingHandler
{
  public LoggingHandler (HttpMessageHandler nextHandler)
  {
     InnerHandler = nextHandler;
  }

  protected async override Task <HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken)
  {
    Console.WriteLine ("Requesting: " + request.RequestUri);
    var response = await base.SendAsync (request, cancellationToken);
    Console.WriteLine ("Got response: " + response.StatusCode);
    return response;
  }
}
```

Observa que hemos mantenido la asincronía al anular SendAsync. Introducir el modificador async al anular un método que devuelve una tarea es perfectamente válido y deseable en este caso.

Una solución mejor que escribir en la consola sería tener en cuenta un objeto de registro en el constructor. Incluso mejor sería aceptar un par de delegados Action<T> que indiquen cómo registrar los objetos de solicitud y respuesta.

Proxies

Un servidor proxy es un intermediario a través del cual se pueden enrutar las solicitudes HTTP. Las organizaciones a veces configuran un servidor proxy como el único medio por el cual los empleados pueden acceder a Internet, principalmente porque simplifica la seguridad. Un proxy tiene una dirección propia y puede exigir autenticación para que solo los usuarios seleccionados en la LAN puedan acceder a Internet.

Para usar un proxy con HttpClient, primero crea un HttpClientHandler y asigna su propiedad Proxy, luego se pasa al constructor de HttpClient:

```
WebProxy p = new WebProxy ("192.178.10.49", 808);
p.Credentials = new NetworkCredential ("username", "password", "domain");

var handler = new HttpClientHandler { Proxy = p };
var client = new HttpClient (handler);
...
```

HttpClientHandler también tiene una propiedad UseProxy que puedes asignar a false en lugar de anular la propiedad Proxy para desactivar la detección automática.

Si proporcionas un dominio al construir NetworkCredential, se utilizarán protocolos de autenticación basados en Windows. Para usar el usuario de Windows autenticado actualmente, asigna el valor estático CredentialCache.DefaultNetworkCredentials a la propiedad Credentials del proxy.

Como alternativa a establecer repetidamente el proxy, puedes establecer el valor predeterminado global de la siguiente manera:

```
HttpClient.DefaultWebProxy = myWebProxy;
```

Autenticación

Puedes proporcionar un nombre de usuario y contraseña a HttpClient de la siguiente manera:

```
string username = "myuser";
string password = "mypassword";

var handler = new HttpClientHandler();
handler.Credentials = new NetworkCredential (username, password);
var client = new HttpClient (handler);
...
```

Esto funciona con protocolos de autenticación basados en diálogos, como Basic y Digest, y es extensible a través de la clase AuthenticationManager. También es compatible con NTLM y Kerberos de Windows (si incluyes un nombre de dominio al construir el objeto NetworkCredential). Si deseas utilizar el usuario de Windows autenticado actualmente, puedes dejar la propiedad Credentials como nula y, en su lugar, establecer UseDefaultCredentials en true.

Cuando proporcionas credenciales, HttpClient negocia automáticamente un protocolo compatible. En algunos casos, puede haber una opción: si examinas la respuesta inicial de una página web de correo de Microsoft Exchange, por ejemplo, puede contener los siguientes encabezados:

```
HTTP/1.1 401 Unauthorized
Content-Length: 83
Content-Type: text/html
Server: Microsoft-IIS/6.0
WWW-Authenticate: Negotiate
WWW-Authenticate: NTLM
WWW-Authenticate: Basic realm="exchange.somedomain.com"
X-Powered-By: ASP.NET
Date: Sat, 05 Aug 2006 12:37:23 GMT
```

El código 401 indica que se requiere autorización; los encabezados "WWW-Authenticate" indican qué protocolos de autenticación se entienden. Sin embargo, si configuras HttpClientHandler con el nombre de usuario y la contraseña correctos, este mensaje se ocultará porque el tiempo de ejecución responde automáticamente eligiendo un protocolo de autenticación compatible y luego reenviando la solicitud original con un encabezado adicional. Aquí tienes un ejemplo:

```
Authorization: Negotiate TlRMTVNTUAAABAAAt5II2gjACDArAAACAwACACgAAAAQ
ATmKAAAAD0lVDRdPUksHUq9VUA==
```

Este mecanismo proporciona transparencia, pero genera un viaje adicional con cada solicitud. Puedes evitar los viajes adicionales en solicitudes posteriores al mismo URI estableciendo la propiedad PreAuthenticate del HttpClientHandler en true.

CredentialCache

Puedes forzar un protocolo de autenticación específico con un objeto CredentialCache. Una caché de credenciales contiene uno o más objetos NetworkCredential, cada uno asociado a un protocolo específico y un prefijo de URI. Por ejemplo, es posible que desees evitar el protocolo Basic al iniciar sesión en un servidor de Exchange porque transmite contraseñas en texto sin formato:

```
CredentialCache cache = new CredentialCache();
Uri prefix = new Uri ("http://exchange.somedomain.com");
cache.Add (prefix, "Digest",  new NetworkCredential ("joe", "passwd"));
cache.Add (prefix, "Negotiate", new NetworkCredential ("joe", "passwd"));

var handler = new HttpClientHandler();
handler.Credentials = cache;
...
```

Se especifica un protocolo de autenticación como una cadena. Los valores válidos incluyen:

```
Basic, Digest, NTLM, Kerberos, Negotiate
```

En esta situación particular, elegirá "Negotiate" porque el servidor no indicó que admitiera "Digest" en sus encabezados de autenticación. "Negotiate" es un protocolo de Windows que actualmente se reduce a Kerberos o NTLM, según las capacidades del servidor, pero garantiza la compatibilidad futura de tu aplicación cuando se implementen futuros estándares de seguridad.

La propiedad estática "CredentialCache.DefaultNetworkCredentials" te permite agregar el usuario de Windows autenticado actualmente a la caché de credenciales sin tener que especificar una contraseña:

```
cache.Add (prefix, "Negotiate", CredentialCache.DefaultNetworkCredentials);
```

Autenticación mediante encabezados

Otra forma de autenticación es establecer directamente el encabezado de autenticación:

```
var client = new HttpClient();
client.DefaultRequestHeaders.Authorization =
  new AuthenticationHeaderValue ("Basic",
    Convert.ToBase64String (Encoding.UTF8.GetBytes ("username:password")));
...
```

Esta estrategia también funciona con sistemas de autenticación personalizados como OAuth.

Encabezados

HttpClient te permite agregar encabezados HTTP personalizados a una solicitud, así como enumerar los encabezados en una respuesta. Un encabezado es simplemente un par clave/valor que contiene metadatos, como el tipo de contenido del mensaje o el software del servidor. HttpClient expone colecciones de tipos fuertemente tipados con propiedades para encabezados HTTP estándar. La propiedad DefaultRequestHeaders es para encabezados que se aplican a cada solicitud:

```
var client = new HttpClient (handler);

client.DefaultRequestHeaders.UserAgent.Add (
  new ProductInfoHeaderValue ("VisualStudio", "2022"));

client.DefaultRequestHeaders.Add ("CustomHeader", "VisualStudio/2022");
```

Sin embargo, la propiedad Headers en la clase HttpRequestMessage es para encabezados específicos de una solicitud.

Cadenas de consulta

Una cadena de consulta es simplemente una cadena que se agrega a un URI con un signo de interrogación y se utiliza para enviar datos simples al servidor. Puedes especificar múltiples pares clave/valor en una cadena de consulta con la siguiente sintaxis:

```
?key1=value1&key2=value2&key3=value3...
```

Aquí tienes un ejemplo de un URI con una cadena de consulta:

```
string requestURI = "http://www.google.com/search?q=HttpClient&hl=fr";
```

Si existe la posibilidad de que tu consulta incluya símbolos o espacios, puedes usar el método EscapeDataString de Uri para crear un URI válido:

```
string search = Uri.EscapeDataString ("(HttpClient or HttpRequestMessage)");
string language = Uri.EscapeDataString ("fr");
string requestURI = "http://www.google.com/search?q=" + search +
                    "&hl=" + language;
```

El URI resultante sería:

```
http://www.google.com/search?q=(HttpClient%20OR%20HttpRequestMessage)&hl=fr
```

(EscapeDataString es similar a EscapeUriString, excepto que también codifica caracteres como & y =, que de lo contrario podrían afectar la cadena de consulta).

Cargar datos de formulario

Para cargar datos de formulario HTML, crea y rellena un objeto FormUrlEncodedContent. Luego, puedes pasarlo al método PostAsync o asignarlo a la propiedad Content de una solicitud:

```
string uri = "http://www.albahari.com/EchoPost.aspx";
var client = new HttpClient();
var dict = new Dictionary<string,string>
{
    { "Name", "Joe Albahari" },
    { "Company", "O'Reilly" }
};
var values = new FormUrlEncodedContent (dict);
var response = await client.PostAsync (uri, values);
response.EnsureSuccessStatusCode();
Console.WriteLine (await response.Content.ReadAsStringAsync());
```

Cookies

Una cookie es un par de cadenas de nombre/valor que un servidor HTTP envía a un cliente en un encabezado de respuesta. Un cliente de navegador web generalmente recuerda las cookies y las envía nuevamente al servidor en cada solicitud subsiguiente (a la misma dirección) hasta que caducan. Una cookie permite que un servidor sepa si está comunicándose con el mismo cliente que hace un minuto o ayer, sin necesidad de una cadena de consulta complicada en el URI.

De forma predeterminada, HttpClient ignora las cookies recibidas del servidor. Para aceptar cookies, crea un objeto CookieContainer y asígnalo a un HttpClientHandler:

```
var cc = new CookieContainer();
var handler = new HttpClientHandler();
handler.CookieContainer = cc;
var client = new HttpClient (handler);
...
```

Para volver a enviar las cookies recibidas en futuras solicitudes, simplemente utiliza el mismo objeto CookieContainer nuevamente. Alternativamente, puedes comenzar con un nuevo CookieContainer y luego agregar cookies manualmente de la siguiente manera:

```
Cookie c = new Cookie ("PREF",
                       "ID=6b10df1da493a9c4:TM=1179...",
                       "/",
                       ".google.com");
freshCookieContainer.Add (c);
```

El tercer y cuarto argumento indican la ruta y el dominio del originador. Un objeto CookieContainer en el cliente puede contener cookies de muchos lugares diferentes; HttpClient solo envía aquellas cookies cuya ruta y dominio coinciden con los del servidor.

Escribiendo un Servidor HTTP



Si necesitas escribir un servidor HTTP en .NET 6, un enfoque alternativo de nivel superior es utilizar el ASP.NET minimal API. Aquí tienes todo lo que necesitas para empezar:

```
var app = WebApplication.CreateBuilder().Build();
app.MapGet ("/", () => "Hello, world!");
app.Run();
```

También puedes escribir tu propio servidor HTTP en .NET utilizando la clase HttpListener. A continuación se muestra un ejemplo de un servidor simple que escucha en el puerto 51111, espera una solicitud de cliente y devuelve una respuesta de una línea:

```
using var server = new SimpleHttpServer();

// Make a client request:
Console.WriteLine (await new HttpClient().GetStringAsync
  ("http://localhost:51111/MyApp/Request.txt"));

class SimpleHttpServer : IDisposable
{
  readonly HttpListener listener = new HttpListener();

  public SimpleHttpServer() => ListenAsync();
  async void ListenAsync()
  {
    listener.Prefixes.Add ("http://localhost:51111/MyApp/");  // Listen on
    listener.Start();                                         // port 51111

    // Await a client request:
    HttpListenerContext context = await listener.GetContextAsync();

    // Respond to the request:
    string msg = "You asked for: " + context.Request.RawUrl;
    context.Response.ContentLength64 = Encoding.UTF8.GetByteCount (msg);
    context.Response.StatusCode = (int)HttpStatusCode.OK;

    using (Stream s = context.Response.OutputStream)
    using (StreamWriter writer = new StreamWriter (s))
      await writer.WriteAsync (msg);
  }

  public void Dispose() => listener.Close();

}

OUTPUT: You asked for: /MyApp/Request.txt
```

En Windows, HttpListener no utiliza internamente objetos Socket de .NET; en su lugar, llama a la API del servidor HTTP de Windows. Esto permite que muchas aplicaciones en una computadora escuchen en la misma dirección IP y puerto, siempre y cuando cada una registre prefijos de dirección diferentes. En nuestro ejemplo, registramos el prefijo http://localhost/myapp, por lo que otra aplicación podría escuchar en la misma dirección IP y puerto utilizando un prefijo diferente, como http://localhost/anotherapp. Esto es valioso porque abrir nuevos puertos en los firewalls corporativos puede ser políticamente difícil.

HttpListener espera la siguiente solicitud de cliente cuando llamas a GetContext, devolviendo un objeto con propiedades de Request (solicitud) y Response (respuesta). Cada una es análoga a la solicitud o respuesta del cliente, pero desde la perspectiva del servidor. Puedes leer y escribir encabezados y cookies, por ejemplo, en los objetos de solicitud y respuesta, de manera similar a como lo harías en el extremo del cliente.

Puedes elegir cuánto quieres soportar las características del protocolo HTTP, según tu audiencia de clientes prevista. Como mínimo, debes establecer la longitud del contenido y el código de estado en cada solicitud.

Aquí tienes un ejemplo muy sencillo de un servidor de páginas web escrito de forma asíncrona:

```
using System;
using System.IO;
using System.Net;
using System.Text;
using System.Threading.Tasks;

class WebServer
{
  HttpListener _listener;
  string _baseFolder;      // Your web page folder.

  public WebServer (string uriPrefix, string baseFolder)
  {
    _listener = new HttpListener();
    _listener.Prefixes.Add (uriPrefix);
    _baseFolder = baseFolder;
  }

  public async void Start()
  {
    _listener.Start();
    while (true)
      try
      {
        var context = await _listener.GetContextAsync();
        Task.Run (() => ProcessRequestAsync (context));
      }
      catch (HttpListenerException)     { break; }   // Listener stopped.
      catch (InvalidOperationException) { break; }   // Listener stopped.
  }

  public void Stop() => _listener.Stop();

  async void ProcessRequestAsync (HttpListenerContext context)
  {
    try
    {
      string filename = Path.GetFileName (context.Request.RawUrl);
      string path = Path.Combine (_baseFolder, filename);
      byte[] msg;
      if (!File.Exists (path))
      {
        Console.WriteLine ("Resource not found: " + path);
        context.Response.StatusCode = (int) HttpStatusCode.NotFound;
        msg = Encoding.UTF8.GetBytes ("Sorry, that page does not exist");
      }
      else
      {
        context.Response.StatusCode = (int) HttpStatusCode.OK;
        msg = File.ReadAllBytes (path);
      }
      context.Response.ContentLength64 = msg.Length;
      using (Stream s = context.Response.OutputStream)
        await s.WriteAsync (msg, 0, msg.Length);
    }
    catch (Exception ex) { Console.WriteLine ("Request error: " + ex); }
  }
}
```

El siguiente código pone las cosas en marcha:

```
// Listen on port 51111, serving files in d:\webroot:
var server = new WebServer ("http://localhost:51111/", @"d:\webroot");
try
{
  server.Start();
  Console.WriteLine ("Server running... press Enter to stop");
  Console.ReadLine();
}
finally { server.Stop(); }
```

Puedes probar esto desde el extremo del cliente con cualquier navegador web; la URI en este caso será http://localhost:51111/ seguido del nombre de la página web.



HttpListener no se iniciará si otro software está compitiendo por el mismo puerto (a menos que ese software también utilice la API del servidor HTTP de Windows). Ejemplos de aplicaciones que podrían escuchar en el puerto predeterminado 80 incluyen un servidor web o un programa peer-to-peer como Skype.

Nuestro uso de funciones asíncronas hace que este servidor sea escalable y eficiente. Sin embargo, iniciar esto desde un hilo de interfaz de usuario (UI) obstaculizaría la escalabilidad, ya que para cada solicitud, la ejecución volvería al hilo de la UI después de cada operación "await". Incurrir en tal sobrecarga es particularmente inútil dado que no tenemos estado compartido, por lo que en un escenario de interfaz de usuario nos desvincularíamos del hilo de la UI, ya sea de esta manera:

```
Task.Run (Start);
```

O llamando a ConfigureAwait(false) después de llamar a GetContextAsync.

Ten en cuenta que utilizamos Task.Run para llamar a ProcessRequestAsync incluso si el método ya es asíncrono. Esto permite que el llamador procese otra solicitud de inmediato en lugar de tener que esperar la fase sincrónica del método (hasta el primer await).

Usando DNS

La clase estática Dns encapsula el DNS, que realiza la conversión entre una dirección IP en bruto, como 66.135.192.87, y un nombre de dominio amigable para el usuario, como ebay.com.

El método GetHostAddresses convierte de nombre de dominio a dirección IP (o direcciones):

```
foreach (IPAddress a in Dns.GetHostAddresses ("albahari.com"))
  Console.WriteLine (a.ToString());     // 205.210.42.167
```

El método GetHostEntry realiza la conversión inversa, de dirección a nombre de dominio:

```
IPHostEntry entry = Dns.GetHostEntry ("205.210.42.167");
Console.WriteLine (entry.HostName);                    // albahari.com
```

GetHostEntry también acepta un objeto IPAddress, por lo que puedes especificar una dirección IP como un arreglo de bytes:

```
IPAddress address = new IPAddress (new byte[] { 205, 210, 42, 167 });
IPHostEntry entry = Dns.GetHostEntry (address);
Console.WriteLine (entry.HostName);                    // albahari.com
```

Los nombres de dominio se resuelven automáticamente a direcciones IP cuando utilizas clases como WebRequest o TcpClient. Sin embargo, si planeas hacer muchas solicitudes de red a la misma dirección durante la vida útil de una aplicación, a veces puedes mejorar el rendimiento utilizando Dns para convertir explícitamente el nombre de dominio en una dirección IP y luego comunicarte directamente con la dirección IP a partir de ese punto. Esto evita repeticiones en la resolución del mismo nombre de dominio y puede ser beneficioso al tratar a nivel de transporte (a través de TcpClient, UdpClient o Socket).

La clase DNS también proporciona métodos asíncronos basados en tareas que se pueden esperar:

```
foreach (IPAddress a in await Dns.GetHostAddressesAsync ("albahari.com"))
  Console.WriteLine (a.ToString());
```

Enviando correo con SmtpClient

La clase SmtpClient en el espacio de nombres System.Net.Mail te permite enviar mensajes de correo a través del protocolo Simple Mail Transfer Protocol (SMTP). Para enviar un mensaje de texto simple, instanciarías SmtpClient, establecerías su propiedad Host con la dirección de tu servidor SMTP y luego llamarías a Send:

```
SmtpClient client = new SmtpClient();
client.Host = "mail.myserver.com";
client.Send ("from@adomain.com", "to@adomain.com", "subject", "body");
```

Al construir un objeto MailMessage, tienes opciones adicionales, como la capacidad de agregar adjuntos:

```
SmtpClient client = new SmtpClient();
client.Host = "mail.myisp.net";
MailMessage mm = new MailMessage();

mm.Sender = new MailAddress ("kay@domain.com", "Kay");
mm.From   = new MailAddress ("kay@domain.com", "Kay");
mm.To.Add  (new MailAddress ("bob@domain.com", "Bob"));
mm.CC.Add  (new MailAddress ("dan@domain.com", "Dan"));
mm.Subject = "Hello!";
mm.Body = "Hi there. Here's the photo!";
mm.IsBodyHtml = false;
mm.Priority = MailPriority.High;

Attachment a = new Attachment ("photo.jpg",
                               System.Net.Mime.MediaTypeNames.Image.Jpeg);
mm.Attachments.Add (a);
client.Send (mm);
```

Para frustrar a los spammers, la mayoría de los servidores SMTP en Internet solo aceptarán conexiones autenticadas y requieren comunicación a través de SSL:

```
var client = new SmtpClient ("smtp.myisp.com", 587)
{
  Credentials = new NetworkCredential ("me@myisp.com", "MySecurePass"),
  EnableSsl = true
};
client.Send ("me@myisp.com", "someone@somewhere.com", "Subject", "Body");
Console.WriteLine ("Sent");
```

Al cambiar la propiedad DeliveryMethod, puedes instruir a SmtpClient para que en su lugar use IIS para enviar mensajes de correo o simplemente escriba cada mensaje en un archivo .eml en un directorio especificado. Esto puede ser útil durante el desarrollo:

```
SmtpClient client = new SmtpClient();
client.DeliveryMethod = SmtpDeliveryMethod.SpecifiedPickupDirectory;
client.PickupDirectoryLocation = @"c:\mail";
```

Usando TCP

TCP y UDP constituyen los protocolos de la capa de transporte sobre los cuales se construyen la mayoría de los servicios de Internet y LAN. HTTP (versión 2 y anteriores), FTP y SMTP utilizan TCP; DNS y HTTP versión 3 utilizan UDP. TCP es orientado a la conexión e incluye mecanismos de confiabilidad; UDP es sin conexión, tiene un menor costo y admite la difusión. BitTorrent utiliza UDP, al igual que Voice over IP (VoIP).

La capa de transporte ofrece una mayor flexibilidad, y potencialmente un rendimiento mejorado, en comparación con las capas superiores, pero requiere que manejes tareas como la autenticación y el cifrado por ti mismo.

En .NET, con TCP, tienes la opción de usar las clases de fachada más fáciles de usar, TcpClient y TcpListener, o la clase Socket, que es más completa (de hecho, puedes combinar ambas, ya que TcpClient expone el objeto Socket subyacente a través de la propiedad Client). La clase Socket expone más opciones de configuración y permite el acceso directo a la capa de red (IP) y a protocolos no basados en Internet, como SPX/IPX de Novell.

Al igual que con otros protocolos, TCP diferencia entre un cliente y un servidor: el cliente inicia una solicitud, mientras que el servidor espera una solicitud. Aquí tienes la estructura básica para una solicitud de cliente TCP síncrona:

```
using (TcpClient client = new TcpClient())
{
  client.Connect ("address", port);
  using (NetworkStream n = client.GetStream())
  {
    // Read and write to the network stream...
  }
}
```

El método Connect de TcpClient bloquea hasta que se establece una conexión (ConnectAsync es la versión asíncrona). Luego, NetworkStream proporciona un medio de comunicación bidireccional, tanto para transmitir como para recibir bytes de datos desde un servidor.

Un servidor TCP simple se vería así:

```
TcpListener listener = new TcpListener (<ip address>, port);
listener.Start();

while (keepProcessingRequests)
  using (TcpClient c = listener.AcceptTcpClient())
  using (NetworkStream n = c.GetStream())
  {
    // Read and write to the network stream...
  }

listener.Stop();
```

TcpListener requiere la dirección IP local en la que escuchar (una computadora con dos tarjetas de red, por ejemplo, puede tener dos direcciones). Puedes usar IPAddress.Any para indicarle que escuche en todas (o la única) dirección IP local. AcceptTcpClient bloquea hasta que se recibe una solicitud de cliente (nuevamente, también hay una versión asíncrona), momento en el que llamamos a GetStream, al igual que en el lado del cliente.

Cuando trabajas en la capa de transporte, debes decidir un protocolo para determinar quién habla cuándo y por cuánto tiempo, similar a un walkie-talkie. ¡Si ambas partes hablan o escuchan al mismo tiempo, la comunicación se interrumpe!

Vamos a inventar un protocolo en el que el cliente hable primero, diciendo "Hola", y luego el servidor responda diciendo "¡Hola de vuelta!". Aquí está el código:

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;

new Thread (Server).Start();       // Run server method concurrently.
Thread.Sleep (500);                // Give server time to start.
Client();

void Client()
{
  using (TcpClient client = new TcpClient ("localhost", 51111))
  using (NetworkStream n = client.GetStream())
  {
    BinaryWriter w = new BinaryWriter (n);
    w.Write ("Hello");
    w.Flush();
    Console.WriteLine (new BinaryReader (n).ReadString());
  }
}

void Server()     // Handles a single client request, then exits.
{
  TcpListener listener = new TcpListener (IPAddress.Any, 51111);
  listener.Start();
  using (TcpClient c = listener.AcceptTcpClient())
  using (NetworkStream n = c.GetStream())
  {
    string msg = new BinaryReader (n).ReadString();
    BinaryWriter w = new BinaryWriter (n);
    w.Write (msg + " right back!");
    w.Flush();                     // Must call Flush because we're not
  }                                // disposing the writer.
  listener.Stop();
}

// OUTPUT: Hello right back!
```

En este ejemplo, estamos utilizando el bucle de retroalimentación local (localhost) para ejecutar el cliente y el servidor en la misma máquina. Hemos elegido arbitrariamente un puerto en el rango no asignado (por encima de 49152) y utilizamos BinaryWriter y BinaryReader para codificar los mensajes de texto. Hemos evitado cerrar o desechar los lectores y escritores para mantener el NetworkStream subyacente abierto hasta que se complete nuestra conversación.

Puede parecer extraño utilizar BinaryReader y BinaryWriter para leer y escribir cadenas de texto. Sin embargo, tienen una ventaja importante sobre StreamReader y StreamWriter: prefijan las cadenas con un entero que indica la longitud, por lo que un BinaryReader siempre sabe exactamente cuántos bytes debe leer. Si llamas a StreamReader.ReadToEnd, podrías bloquearte indefinidamente, ¡porque un NetworkStream no tiene un final definido! Mientras la conexión esté abierta, el flujo de red nunca puede estar seguro de que el cliente no enviará más datos.



De hecho, StreamReader está completamente fuera de límites con NetworkStream, incluso si solo planeas llamar a ReadLine. Esto se debe a que StreamReader tiene un búfer de lectura anticipada, lo que puede resultar en que lea más bytes de los que están disponibles actualmente, bloqueándose indefinidamente (o hasta que el socket agote el tiempo de espera). Otros flujos, como FileStream, no sufren esta incompatibilidad con StreamReader porque tienen un fin definido, momento en el que Read devuelve inmediatamente un valor de 0.

Concurrencia con TCP

TcpClient y TcpListener ofrecen métodos asíncronos basados en tareas para lograr una concurrencia escalable. Utilizar estos métodos es simplemente cuestión de reemplazar las llamadas a métodos bloqueantes por sus versiones *Async y esperar la tarea que se devuelve.

En el siguiente ejemplo, escribimos un servidor TCP asíncrono que acepta solicitudes de 5,000 bytes de longitud, invierte los bytes y luego los envía de vuelta al cliente:

```
async void RunServerAsync ()
{
  var listener = new TcpListener (IPAddress.Any, 51111);
  listener.Start ();
  try
  {
    while (true)
      Accept (await listener.AcceptTcpClientAsync ());
  }
  finally { listener.Stop(); }
}

async Task Accept (TcpClient client)
{
  await Task.Yield ();
  try
  {
    using (client)
    using (NetworkStream n = client.GetStream ())
    {
      byte[] data = new byte [5000];

      int bytesRead = 0; int chunkSize = 1;
      while (bytesRead < data.Length && chunkSize > 0)
        bytesRead += chunkSize =
          await n.ReadAsync (data, bytesRead, data.Length - bytesRead);

      Array.Reverse (data);   // Reverse the byte sequence
      await n.WriteAsync (data, 0, data.Length);
    }
  }

  catch (Exception ex) { Console.WriteLine (ex.Message); }
}
```

Un programa como este es escalable en el sentido de que no bloquea un hilo durante la duración de una solicitud. Por lo tanto, si 1,000 clientes se conectaran al mismo tiempo a través de una conexión de red lenta (de manera que cada solicitud tomara varios segundos de principio a fin, por ejemplo), este programa no requeriría 1,000 hilos durante ese tiempo (a diferencia de una solución sincrónica). En su lugar, arrienda hilos solo durante los breves períodos de tiempo requeridos para ejecutar el código antes y después de las expresiones await.

Recibiendo correos POP3 con TCP

.NET no proporciona soporte a nivel de aplicación para POP3, por lo que es necesario escribir en el nivel TCP para recibir correos de un servidor POP3. Afortunadamente, se trata de un protocolo simple; una conversación POP3 se desarrolla de la siguiente manera:

Client	Mail server	Notes
Client connects...	`+OK Hello there.`	Welcome message
`USER joe`	`+OK Password required.`	
`PASS password`	`+OK Logged in.`	

`LIST`	`+OK`
	`1 1876`
	`2 5412`
	`3 845`
	`.`
		Lists the ID and file size of each message on the server

`RETR 1`	`+OK 1876 octets`
	Content of message #1...
	`.`
		Retrieves the message with the specified ID

`DELE 1`	`+OK Deleted.`	Deletes a message from the server
`QUIT`	`+OK Bye-bye.`	

Cada comando y respuesta se termina con una nueva línea (CR + LF), excepto para los comandos LIST y RETR de varias líneas, que se terminan con un solo punto en una línea separada. Debido a que no podemos usar StreamReader con NetworkStream, podemos comenzar escribiendo un método auxiliar para leer una línea de texto de manera no almacenada en búfer:

```
string ReadLine (Stream s)
{
  List<byte> lineBuffer = new List<byte>();
  while (true)
  {
    int b = s.ReadByte();
    if (b == 10 || b < 0) break;
    if (b != 13) lineBuffer.Add ((byte)b);
  }
  return Encoding.UTF8.GetString (lineBuffer.ToArray());
}
```

También necesitamos un método auxiliar para enviar un comando. Dado que siempre esperamos recibir una respuesta que comienza con "+OK", podemos leer y validar la respuesta al mismo tiempo:

```
void SendCommand (Stream stream, string line)
{
  byte[] data = Encoding.UTF8.GetBytes (line + "\r\n");
  stream.Write (data, 0, data.Length);
  string response = ReadLine (stream);
  if (!response.StartsWith ("+OK"))
    throw new Exception ("POP Error: " + response);
}
```

Con estos métodos escritos, la tarea de recuperar correos electrónicos es sencilla. Establecemos una conexión TCP en el puerto 110 (el puerto POP3 predeterminado) y luego comenzamos a comunicarnos con el servidor. En este ejemplo, escribimos cada mensaje de correo en un archivo con un nombre aleatorio y extensión .eml, antes de eliminar el mensaje del servidor:

```
using (TcpClient client = new TcpClient ("mail.isp.com", 110))
using (NetworkStream n = client.GetStream())
{
  ReadLine (n);                              // Read the welcome message.
  SendCommand (n, "USER username");
  SendCommand (n, "PASS password");
  SendCommand (n, "LIST");                   // Retrieve message IDs
  List<int> messageIDs = new List<int>();
  while (true)
  {
    string line = ReadLine (n);              // e.g.,  "1 1876"
    if (line == ".") break;
    messageIDs.Add (int.Parse (line.Split (' ')[0] ));   // Message ID
  }

  foreach (int id in messageIDs)         // Retrieve each message.
  {
    SendCommand (n, "RETR " + id);
    string randomFile = Guid.NewGuid().ToString() + ".eml";
    using (StreamWriter writer = File.CreateText (randomFile))
      while (true)
      {
        string line = ReadLine (n);      // Read next line of message.
        if (line == ".") break;          // Single dot = end of message.
        if (line == "..") line = ".";    // "Escape out" double dot.
        writer.WriteLine (line);         // Write to output file.
      }
    SendCommand (n, "DELE " + id);       // Delete message off server.
  }
  SendCommand (n, "QUIT");
}
```



Puedes encontrar bibliotecas POP3 de código abierto en NuGet que brindan soporte para aspectos del protocolo como autenticación, conexiones TLS/SSL, análisis MIME y más.

17 Ensamblados (Assemblies)

Un ensamblado es la unidad básica de implementación en .NET y también el contenedor de todos los tipos. Un ensamblado contiene tipos compilados con su código de Lenguaje Intermedio (IL), recursos en tiempo de ejecución e información para ayudar con la versión y la referencia a otros ensamblados. Un ensamblado también define un límite para la resolución de tipos. En .NET, un ensamblado consta de un único archivo con una extensión .dll.



Cuando construyes una aplicación ejecutable en .NET, terminas con dos archivos: un ensamblado (.dll) y un lanzador ejecutable (.exe) apropiado para la plataforma a la que estás apuntando. Esto difiere de lo que ocurre en .NET Framework, que genera un ensamblado ejecutable portable (PE). Un PE tiene una extensión .exe y actúa tanto como un ensamblado como un lanzador de aplicaciones. Un PE puede dirigirse simultáneamente a versiones de 32 y 64 bits de Windows.

La mayoría de los tipos en este capítulo provienen de los siguientes espacios de nombres:

```
System.Reflection          System.Reflection
System.Resources           System.Resources
System.Globalization       System.Globalization
```

Qué contiene un ensamblado

Un ensamblado contiene cuatro tipos de elementos:

Manifiesto del ensamblado

> proporciona información al CLR (Common Language Runtime), como el nombre del ensamblado, su versión y otros ensamblados a los que hace referencia.

Manifiesto de la aplicación

> proporciona información al sistema operativo sobre cómo se debe implementar el ensamblado y si se requiere elevación administrativa.

Tipos compilados

> el código IL (Intermediate Language) compilado y los metadatos de los tipos definidos dentro del ensamblado.

Recursos

> otros datos incrustados en el ensamblado, como imágenes y texto localizable.

De estos, solo el manifiesto del ensamblado es obligatorio, aunque un ensamblado casi siempre contiene tipos compilados (a menos que sea un ensamblado de recursos).

El manifiesto del ensamblado

El manifiesto del ensamblado tiene dos propósitos:

- Describe el ensamblado al entorno de ejecución administrado (CLR).

- Actúa como un directorio para los módulos, tipos y recursos en el ensamblado.

Los ensamblados son autosuficientes en cuanto a su descripción. Un consumidor puede descubrir todos los datos, tipos y funciones de un ensamblado sin necesidad de archivos adicionales.

 El manifiesto del ensamblado no se agrega explícitamente a un ensamblado, sino que se incrusta automáticamente en el ensamblado como parte de la compilación.

Aquí hay un resumen de los datos funcionalmente significativos almacenados en el manifiesto:

- El nombre simple del ensamblado.
- Un número de versión (AssemblyVersion).
- Una clave pública y un hash firmado del ensamblado, si está firmado digitalmente (strongly named).
- Una lista de ensamblados referenciados, incluyendo su versión y clave pública.
- Una lista de tipos definidos en el ensamblado.
- La cultura que se apunta, en caso de ser un ensamblado satélite (AssemblyCulture).

El manifiesto también puede almacenar los siguientes datos informativos:

- Un título y descripción completos (AssemblyTitle y AssemblyDescription).
- Información de la empresa y derechos de autor (AssemblyCompany y AssemblyCopyright).
- Una versión informativa para mostrar (AssemblyInformationalVersion).
- Atributos adicionales para datos personalizados.

Algunos de estos datos se derivan de los argumentos proporcionados al compilador, como la lista de ensamblados referenciados o la clave pública con la que se firma el ensamblado. El resto proviene de atributos de ensamblado, indicados entre paréntesis.

 Puedes ver el contenido del manifiesto de un ensamblado con la herramienta de .NET ildasm.exe. En el Capítulo 18, describimos cómo hacerlo programáticamente utilizando reflexión.

Especificación de atributos de ensamblado

Los atributos de ensamblado comúnmente utilizados se pueden especificar en Visual Studio en la página de Propiedades del proyecto, en la pestaña Package. La configuración en esa pestaña se agrega al archivo de proyecto (.csproj).

Para especificar atributos que no son compatibles con la pestaña Package, o si no estás trabajando con un archivo .csproj, puedes especificar los atributos de ensamblado en el código fuente (esto se hace a menudo en un archivo llamado AssemblyInfo.cs).

Un archivo de atributos dedicado contiene solo declaraciones de using y atributos de ensamblado. Por ejemplo, para exponer tipos con ámbito interno a un proyecto de prueba unitaria, harías lo siguiente:

```
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo("MyUnitTestProject")]
```

El Manifiesto de la Aplicación (Windows)

Un manifiesto de aplicación es un archivo XML que comunica información sobre el ensamblado al sistema operativo. Un manifiesto de aplicación se incrusta en el ejecutable de inicio como un recurso Win32 durante el proceso de compilación. Si está presente, el manifiesto se lee y procesa antes de que el CLR cargue el ensamblado y puede influir en cómo Windows inicia el proceso de la aplicación.

Un manifiesto de aplicación .NET tiene un elemento raíz llamado "assembly" en el espacio de nombres XML "urn:schemas-microsoft-com:asm.v1":

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <!-- contents of manifest -->
</assembly>
```

El siguiente manifiesto indica al sistema operativo que solicite elevación administrativa:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="requireAdministrator" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

(Las aplicaciones UWP tienen un manifiesto mucho más elaborado, descrito en el archivo Package.appxmanifest. Esto incluye una declaración de las capacidades del programa, que determinan los permisos otorgados por el sistema operativo. La forma más sencilla de editar este archivo es mediante Visual Studio, que muestra un diálogo cuando se hace doble clic en el archivo del manifiesto.)

Desplegar un manifiesto de aplicación

Puedes agregar un manifiesto de aplicación a un proyecto .NET en Visual Studio haciendo clic derecho en tu proyecto en el Explorador de soluciones, seleccionando "Agregar", luego "Nuevo

elemento" y eligiendo "Archivo de manifiesto de aplicación". Al compilar, el manifiesto se incrustará en el ensamblado de salida.



La herramienta de .NET, ildasm.exe, no reconoce la presencia de un manifiesto de aplicación incrustado. Sin embargo, Visual Studio indica si hay un manifiesto de aplicación incrustado si haces doble clic en el ensamblado en el Explorador de soluciones.

Módulos

El contenido de un ensamblado en realidad se empaqueta dentro de un contenedor intermedio llamado módulo. Un módulo corresponde a un archivo que contiene el contenido de un ensamblado. La razón de esta capa adicional de contenedores es permitir que un ensamblado abarque varios archivos, una característica presente en .NET Framework pero ausente en .NET 5 y .NET Core. La Figura 17-1 ilustra la relación.

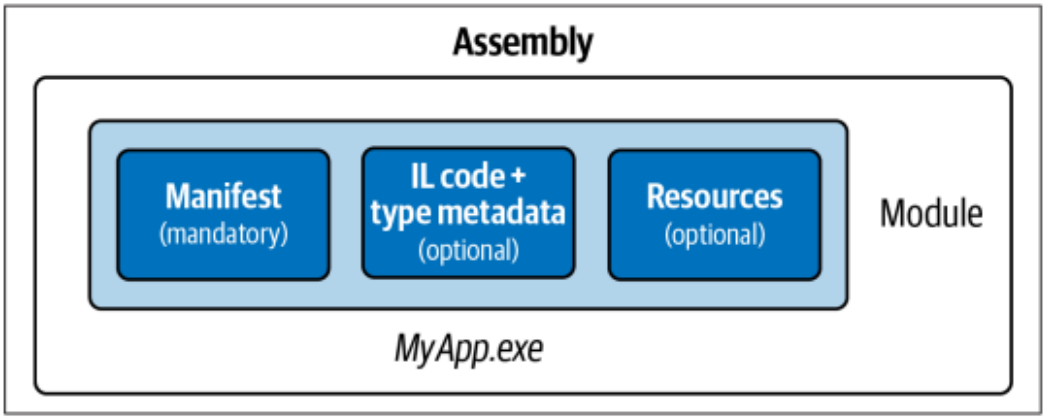


Figure 17-1. Single-file assembly

Aunque .NET no admite ensamblados multifile, en ocasiones es necesario tener en cuenta el nivel adicional de contenedores que imponen los módulos. El escenario principal es con la reflexión (ver "Reflexionando ensamblados" en la página 802 y "Emisión de ensamblados y tipos" en la página 815).

Clase Assembly

La clase Assembly en System.Reflection es una puerta de acceso para acceder a los metadatos de un ensamblado en tiempo de ejecución. Hay varias formas de obtener un objeto de ensamblado: la forma más sencilla es a través de la propiedad Assembly de un Type:

```
Assembly a = typeof (Program).Assembly;
```

También puedes obtener un objeto Assembly llamando a uno de los métodos estáticos de Assembly:

GetExecutingAssembly

Devuelve el ensamblado del tipo que define la función que se está ejecutando actualmente.

GetCallingAssembly

Hace lo mismo que GetExecutingAssembly, pero para la función que llamó a la función que se está ejecutando actualmente.

GetEntryAssembly

Devuelve el ensamblado que define el método de entrada original de la aplicación.

Después de obtener un objeto Assembly, puedes usar sus propiedades y métodos para consultar los metadatos del ensamblado y reflexionar sobre sus tipos. La Tabla 17-1 muestra un resumen de estas funciones.

Table 17-1. Assembly members

Functions	Purpose	See the section...
`FullName`, `GetName`	Returns the fully qualified name or an `AssemblyName` object	"Assembly Names" on page 747
`CodeBase`, `Location`	Location of the assembly file	"Loading, Resolving, and Isolating Assemblies" on page 759
`Load`, `LoadFrom`, `LoadFile`	Manually loads an assembly into memory	"Loading, Resolving, and Isolating Assemblies" on page 759
`GetSatelliteAssembly`	Locates the satellite assembly of a given culture	"Resources and Satellite Assemblies" on page 752
`GetType`, `GetTypes`	Returns a type, or all types, defined in the assembly	"Reflecting and Activating Types" on page 782
`EntryPoint`	Returns the application's entry method, as a `MethodInfo`	"Reflecting and Invoking Members" on page 789
`GetModule`, `GetModules`, `ManifestModule`	Returns all modules, or the main module, of an assembly	"Reflecting Assemblies" on page 802
`GetCustomAttribute`, `GetCustomAttributes`	Returns the assembly's attributes	"Working with Attributes" on page 803

Las firmas fuertes y la firma de ensamblados



La asignación de una firma fuerte a un ensamblado era importante en .NET Framework por dos razones:

Permitía cargar el ensamblado en la "caché global de ensamblados".

Permitía que otros ensamblados con firma fuerte lo referenciaran.

La asignación de una firma fuerte es menos importante en .NET 5 y .NET Core, ya que estas plataformas no tienen una caché global de ensamblados y no imponen la segunda restricción.

Un ensamblado con firma fuerte tiene una identidad única. Esto se logra añadiendo dos elementos de metadatos al manifiesto:

- Un número único que pertenece a los autores del ensamblado.
- Un hash firmado del ensamblado, que demuestra que el titular del número único produjo el ensamblado.

Esto requiere un par de claves pública/privada. La clave pública proporciona el número único de identificación y la clave privada facilita la firma.


La firma de una firma fuerte no es lo mismo que la firma de Authenticode. Hablaremos sobre Authenticode más adelante en este capítulo.

La clave pública es valiosa para garantizar la singularidad de las referencias al ensamblado: un ensamblado con firma fuerte incorpora la clave pública en su identidad.

En .NET Framework, la clave privada protege el ensamblado de manipulaciones, ya que sin la clave privada, nadie puede lanzar una versión modificada del ensamblado sin que la firma se rompa. En la práctica, esto es útil al cargar un ensamblado en la caché global de ensamblados de .NET Framework. En .NET 5 y .NET Core, la firma tiene poco uso porque nunca se verifica.

Agregar una firma fuerte a un ensamblado previamente "sin firma" cambia su identidad. Por esta razón, es recomendable asignar una firma fuerte desde el principio si se cree que el ensamblado podría necesitar una firma fuerte en el futuro.

Cómo asignar una firma fuerte a un ensamblado

Para asignar una firma fuerte a un ensamblado, primero se genera un par de claves pública/privada con la utilidad sn.exe:

```
sn.exe -k MyKeyPair.snk
```


Visual Studio instala un acceso directo llamado "Developer Command Prompt for VS" que abre un símbolo del sistema cuya ruta (PATH) contiene herramientas de desarrollo como sn.exe.

Para generar un nuevo par de claves y guardarlo en un archivo llamado MyKeyPair.snk, puedes utilizar esta herramienta. Si llegas a perder este archivo, perderás permanentemente la capacidad de volver a compilar tu ensamblado con la misma identidad.

Puedes firmar un ensamblado con este archivo actualizando el archivo de proyecto. Desde Visual Studio, ve a la ventana Propiedades del proyecto y luego, en la pestaña "Firma" (Signing), selecciona la casilla "Firmar el ensamblado" y selecciona tu archivo .snk.

El mismo par de claves puede firmar varios ensamblados y aún así tendrán identidades distintas si sus nombres simples difieren.

Nombres de ensamblados

La "identidad" de un ensamblado está compuesta por cuatro elementos de metadatos de su manifiesto:

- Su nombre simple.
- Su versión ("0.0.0.0" si no está presente).
- Su cultura ("neutral" si no es un ensamblado satélite).
- Su token de clave pública ("null" si no tiene una firma fuerte).

El nombre simple no proviene de ningún atributo, sino del nombre del archivo al que fue compilado originalmente (sin ninguna extensión). Por lo tanto, el nombre simple del ensamblado System.Xml.dll es "System.Xml". Cambiar el nombre del archivo no modifica el nombre simple del ensamblado.

El número de versión se obtiene del atributo AssemblyVersion. Es una cadena dividida en cuatro partes de la siguiente manera:

```
major.minor.build.revision
```

Puedes especificar un número de versión de la siguiente manera:

```
[assembly: AssemblyVersion ("2.5.6.7")]
```

La cultura se obtiene del atributo AssemblyCulture y se aplica a los ensamblados satélite, los cuales se describen más adelante en la sección "Recursos y ensamblados satélite" en la página 752.

El token de clave pública se obtiene de la firma fuerte suministrada en tiempo de compilación, como se discutió en la sección anterior.

Nombres completamente calificados

Un nombre completamente calificado de un ensamblado es una cadena que incluye los cuatro componentes identificadores, en el siguiente formato:

```
simple-name, Version=version, Culture=culture, PublicKeyToken=public-key
```

Por ejemplo, el nombre completamente calificado de System.Private.CoreLib.dll es System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e.

Si el ensamblado no tiene el atributo AssemblyVersion, la versión aparecerá como 0.0.0.0. Si no está firmado, su token de clave pública aparecerá como null.

La propiedad FullName de un objeto Assembly devuelve su nombre completamente calificado. El compilador siempre utiliza nombres completamente calificados al registrar las referencias de ensamblado en el manifiesto.



Un nombre completamente calificado de un ensamblado no incluye la ruta del directorio para ayudar a localizarlo en el disco. La localización de un ensamblado que reside en otro directorio es un asunto completamente separado que abordaremos en "Carga, resolución y aislamiento de ensamblados" en la página 759.

La clase AssemblyName

AssemblyName es una clase con una propiedad de tipo para cada uno de los cuatro componentes de un nombre completamente calificado de ensamblado. AssemblyName tiene dos propósitos:

- Analiza o construye un nombre completamente calificado de ensamblado.
- Almacena datos adicionales para ayudar en la resolución (búsqueda) del ensamblado.

Puedes obtener un objeto AssemblyName de las siguientes maneras:

- Instanciando un AssemblyName y proporcionando un nombre completamente calificado.
- Llamando a GetName en un objeto Assembly existente.
- Llamando a AssemblyName.GetAssemblyName y proporcionando la ruta de un archivo de ensamblado en disco.

También puedes instanciar un objeto AssemblyName sin argumentos y luego establecer cada una de sus propiedades para construir un nombre completamente calificado. Un AssemblyName es mutable cuando se construye de esta manera.

Aquí están sus propiedades y métodos esenciales:

```
string      FullName    { get; }           // Fully qualified name
string      Name        { get; set; }      // Simple name
Version     Version     { get; set; }      // Assembly version
CultureInfo CultureInfo { get; set; }      // For satellite assemblies
string      CodeBase    { get; set; }      // Location

byte[]      GetPublicKey();                // 160 bytes
void        SetPublicKey (byte[] key);
byte[]      GetPublicKeyToken();           // 8-byte version
void        SetPublicKeyToken (byte[] publicKeyToken);
```

La propiedad "Version" es una representación fuertemente tipada en sí misma, con propiedades para los números "Major" (mayor), "Minor" (menor), "Build" (compilación) y "Revision" (revisión). El método "GetPublicKey" devuelve la clave pública criptográfica completa, mientras

que el método "GetPublicKeyToken" devuelve los últimos ocho bytes utilizados para establecer la identidad.

Para obtener el nombre simple de un ensamblado utilizando "AssemblyName":

```
Console.WriteLine (typeof (string).Assembly.GetName().Name);
// System.Private.CoreLib
```

Para obtener la versión de un ensamblado:

```
string v = myAssembly.GetName().Version.ToString();
```

Versiones informativas y de archivo del ensamblado

Hay dos atributos adicionales disponibles para expresar información relacionada con la versión. A diferencia de "AssemblyVersion", estos dos atributos no afectan la identidad de un ensamblado y, por lo tanto, no tienen ningún efecto en lo que sucede en tiempo de compilación o en tiempo de ejecución:

AssemblyInformationalVersion

> La versión que se muestra al usuario final. Es visible en el cuadro de diálogo de propiedades del archivo de Windows como "Product Version" (Versión del producto). Aquí se puede incluir cualquier cadena, como "5.1 Beta 2". Típicamente, todos los ensamblados en una aplicación tendrían asignado el mismo número de versión informativa.

AssemblyFileVersion

> Esto se refiere al número de compilación para ese ensamblado. Es visible en el cuadro de diálogo de propiedades del archivo de Windows como "File Version" (Versión del archivo). Al igual que "AssemblyVersion", debe contener una cadena formada por hasta cuatro números separados por puntos.

Firma de Authenticode

Authenticode es un sistema de firma de código cuyo propósito es demostrar la identidad del editor. La firma de Authenticode y la firma de strong-name son independientes: puedes firmar un ensamblado con uno o ambos sistemas.

Aunque la firma de strong-name puede demostrar que los ensamblados A, B y C provienen de la misma parte (asumiendo que la clave privada no se ha filtrado), no puede decirte quién es esa parte. Para saber que la parte era Joe Albahari o Microsoft Corporation, necesitas Authenticode.

Authenticode es útil al descargar programas de Internet, ya que proporciona la garantía de que un programa proviene de quien fue nombrado por la Autoridad de Certificación y no se ha

modificado en tránsito. También evita la advertencia de "Editor desconocido" al ejecutar una aplicación descargada por primera vez.

La firma de Authenticode también es un requisito al enviar aplicaciones a la Tienda de Windows.

Authenticode funciona no solo con ensamblados de .NET, sino también con ejecutables y archivos binarios sin gestionar, como archivos de implementación .msi. Por supuesto, Authenticode no garantiza que un programa esté libre de malware, aunque reduce la probabilidad.

Una persona o entidad ha estado dispuesta a poner su nombre (respaldado por un pasaporte o documento de la empresa) detrás del ejecutable o biblioteca.



El CLR no trata la firma de Authenticode como parte de la identidad de un ensamblado. Sin embargo, puede leer y validar firmas de Authenticode a pedido, como veremos pronto.

Firmar con Authenticode requiere que contactes a una Autoridad de Certificación (CA) con evidencia de tu identidad personal o la identidad de tu empresa (estatutos de constitución, etc.). Después de que la CA haya verificado tus documentos, emitirá un certificado de firma de código X.509 que normalmente es válido de uno a cinco años. Esto te permite firmar ensamblados con la utilidad signtool. También puedes crear tu propio certificado con la utilidad makecert; sin embargo, solo será reconocido en las computadoras en las que el certificado esté instalado explícitamente.

El hecho de que los certificados (no autofirmados) puedan funcionar en cualquier computadora se basa en la infraestructura de clave pública. Esencialmente, tu certificado está firmado con otro certificado perteneciente a una CA. La CA es confiable porque todas las CAs se cargan en el sistema operativo. (Para verlas, ve al Panel de control de Windows y, en el cuadro de búsqueda, escribe "certificado". En la sección Herramientas administrativas, haz clic en "Administrar certificados de computadora". Esto abre el Administrador de certificados. Abre el nodo "Autoridades de certificación raíz confiables" y haz clic en Certificados). Una CA puede revocar el certificado de un editor si se filtra, por lo que verificar una firma de Authenticode requiere preguntar periódicamente a la CA una lista actualizada de revocaciones de certificados.

Debido a que Authenticode utiliza firmas criptográficas, una firma de Authenticode es inválida si alguien posteriormente manipula el archivo. Discutiremos criptografía, hash y firmas en el Capítulo 20.

Cómo firmar con Authenticode

Obtener e instalar un certificado

El primer paso es obtener un certificado de firma de código de una CA (consulta la información adicional a continuación). Luego, puedes trabajar con el certificado como un archivo protegido por contraseña o cargar el certificado en el almacén de certificados de la computadora. La ventaja de hacer lo último es que puedes firmar sin necesidad de especificar una contraseña. Esto es ventajoso porque evita que una contraseña sea visible en scripts de compilación automatizados o archivos por lotes.

Dónde obtener un certificado de firma de código

Solo un puñado de autoridades de certificación (CA) de firma de código están precargadas en Windows como autoridades de certificación raíz. Estas incluyen Comodo, GoDaddy, GlobalSign, DigiCert, Thawte y Symantec.

También existen revendedores como K Software que ofrecen certificados de firma de código con descuento de las autoridades mencionadas anteriormente.

Los certificados de Authenticode emitidos por K Software, Comodo, GoDaddy y GlobalSign se promocionan como menos restrictivos, ya que también firman programas que no son de Microsoft. Aparte de esto, los productos de todos los proveedores son funcionalmente equivalentes.

Cabe destacar que un certificado SSL generalmente no se puede utilizar para la firma de Authenticode (a pesar de utilizar la misma infraestructura X.509). Esto se debe, en parte, a que un certificado SSL se trata de demostrar la propiedad de un dominio, mientras que Authenticode se trata de demostrar quién eres.

Para cargar un certificado en el almacén de certificados de la computadora, abre el Administrador de certificados como se describió anteriormente. Abre la carpeta Personal, haz clic derecho en la carpeta Certificates y selecciona All Tasks/Import. Un asistente de importación te guiará a través del proceso.

Una vez que la importación esté completa, haz clic en el botón View en el certificado, ve a la pestaña Details y copia la huella digital del certificado. Esta es el hash SHA-256 que necesitarás posteriormente para identificar el certificado al firmar.

Si también deseas firmar con strong-name tu ensamblado, debes hacerlo antes de firmar con Authenticode. Esto se debe a que el CLR conoce la firma con Authenticode, pero no viceversa. Entonces, si firmas con strong-name un ensamblado después de haberlo firmado con Authenticode, este último considerará que la adición de la firma con strong-name por parte del CLR es una modificación no autorizada y considerará que el ensamblado ha sido modificado.

Firma con signtool.exe

Puedes firmar tus programas con Authenticode utilizando la utilidad signtool que viene con Visual Studio (búscala en la carpeta Microsoft SDKs\ClickOnce\SignTool dentro de Program Files). El siguiente comando firma un archivo llamado LINQPad.exe con el certificado ubicado en el almacén My Store de la computadora llamado "Joseph Albahari", utilizando el algoritmo seguro de hash SHA256:

```
signtool sign /n "Joseph Albahari" /fd sha256 LINQPad.exe
```

También puedes especificar una descripción y una URL del producto utilizando las opciones /d y /du:

```
... /d LINQPad /du http://www.linqpad.net
```

En la mayoría de los casos, también querrás especificar un servidor de sellado de tiempo (time-stamping server).

Estampado de tiempo

Después de que tu certificado expire, ya no podrás firmar programas. Sin embargo, los programas que hayas firmado antes de su vencimiento seguirán siendo válidos si especificaste un servidor de sellado de tiempo con el interruptor /tr al firmar. La autoridad de certificación te proporcionará una URI (Identificador Uniforme de Recursos) para este propósito. A continuación se muestra un ejemplo para Comodo (o K Software):

```
... /tr http://timestamp.comodoca.com/authenticode /td SHA256
```

Verificación de una firma de Authenticode en un programa

La forma más sencilla de ver una firma de Authenticode en un archivo es ver las propiedades del archivo en el Explorador de Windows (consulta la pestaña Firmas digitales). La utilidad signtool también proporciona una opción para esto.

Recursos y ensamblados satélite

Una aplicación típicamente contiene no solo código ejecutable, sino también contenido como texto, imágenes o archivos XML. Dicho contenido se puede representar en un ensamblado a través de un recurso (resource). Existen dos casos de uso superpuestos para los recursos:

- Incorporar datos que no se pueden incluir en el código fuente, como imágenes.
- Almacenar datos que puedan necesitar traducción en una aplicación multilingüe.

Un recurso de un ensamblado es, en última instancia, una secuencia de bytes con un nombre. Puedes pensar en un ensamblado como un diccionario de matrices de bytes indexadas por cadenas. Esto se puede observar en ildasm si desensamblas un ensamblado que contiene un recurso llamado banner.jpg y otro recurso llamado data.xml:

```
.mresource public banner.jpg
{
  // Offset: 0x00000F58 Length: 0x000004F6
}
.mresource public data.xml
{
  // Offset: 0x00001458 Length: 0x0000027E
}
```

En este caso, banner.jpg y data.xml se incluyeron directamente en el ensamblado, cada uno como su propio recurso incrustado. Esta es la forma más sencilla de trabajar.

.NET también te permite agregar contenido a través de contenedores .resources intermedios. Estos están diseñados para contener contenido que pueda requerir traducción a diferentes idiomas. Los .resources localizados se pueden empaquetar como ensamblados satélite individuales que se recogen automáticamente en tiempo de ejecución según el idioma del sistema operativo del usuario.

La Figura 17-2 ilustra un ensamblado que contiene dos recursos incrustados directamente, además de un contenedor .resources llamado welcome.resources, para el cual hemos creado dos ensamblados satélites localizados.

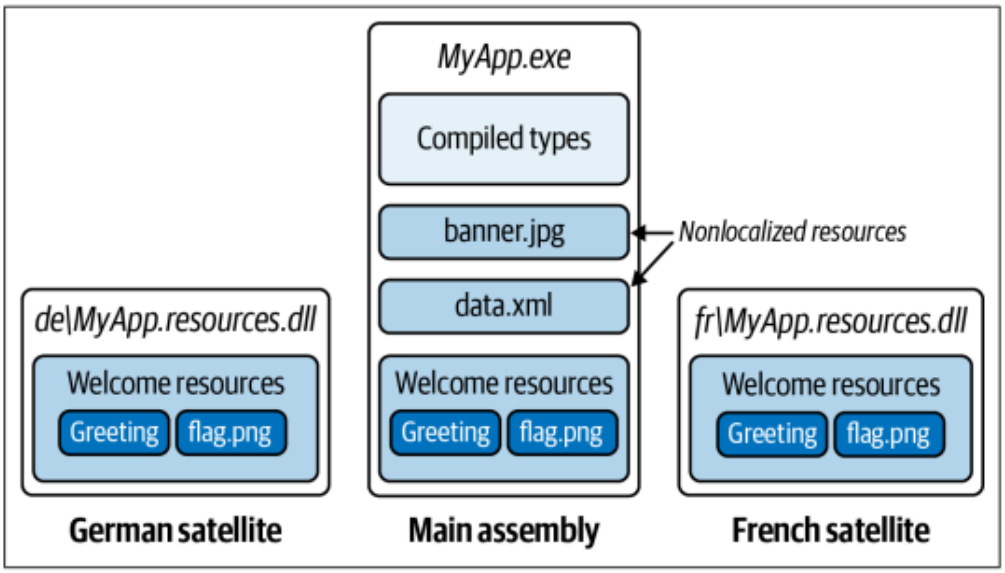


Figure 17-2. Resources

Incrustación Directa de Recursos (Directly Embedding Resources)



La incrustación directa de recursos en los ensamblados no es compatible con las aplicaciones de Windows Store. En su lugar, debes agregar cualquier archivo adicional a tu paquete de implementación y acceder a ellos mediante la lectura desde la carpeta de almacenamiento de tu aplicación (Package.Current.InstalledLocation).

Para incrustar directamente un recurso utilizando Visual Studio:

- Agrega el archivo a tu proyecto.
- Establece su acción de compilación (build action) como "Embedded Resource".

Visual Studio siempre agrega un prefijo al nombre del recurso que consiste en el espacio de nombres predeterminado del proyecto, seguido de los nombres de las subcarpetas en las que se encuentra el archivo. Por ejemplo, si el espacio de nombres predeterminado del proyecto es "Westwind.Reports" y el archivo se llama "banner.jpg" en la carpeta "pictures", el nombre del recurso sería "Westwind.Reports.pictures.banner.jpg".



Los nombres de los recursos distinguen entre mayúsculas y minúsculas. Esto hace que los nombres de las subcarpetas del proyecto en Visual Studio que contienen recursos sean efectivamente sensibles a mayúsculas y minúsculas.

Para recuperar un recurso, debes llamar a GetManifestResourceStream en el ensamblado que contiene el recurso. Esto devuelve un flujo (stream), que luego puedes leer como cualquier otro:

```
Assembly a = Assembly.GetEntryAssembly();

using (Stream s = a.GetManifestResourceStream ("TestProject.data.xml"))
using (XmlReader r = XmlReader.Create (s))
  ...

System.Drawing.Image image;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
  image = System.Drawing.Image.FromStream (s);
```

El flujo de datos devuelto es ajustable, por lo que también puedes hacer lo siguiente:

```
byte[] data;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
  data = new BinaryReader (s).ReadBytes ((int) s.Length);
```

Si has utilizado Visual Studio para incrustar el recurso, debes recordar incluir el prefijo basado en el espacio de nombres. Para evitar errores, puedes especificar el prefijo en un argumento separado utilizando un tipo. El espacio de nombres del tipo se utiliza como prefijo:

```
using (Stream s = a.GetManifestResourceStream (typeof (X), "data.xml"))
```

X puede ser cualquier tipo con el espacio de nombres deseado de tu recurso (normalmente, un tipo en la misma carpeta del proyecto).



Establecer la acción de compilación de un elemento del proyecto en Visual Studio como "Resource" dentro de una aplicación de Windows Presentation Foundation (WPF) no es lo mismo que establecer su acción de compilación como "Embedded Resource". Lo primero añade realmente el elemento a un archivo .resources llamado <AssemblyName>.g.resources, cuyo contenido se accede a través de la clase Application de WPF, utilizando una URI como clave.

Para añadir a la confusión, WPF también sobrecarga el término "recurso". Los recursos estáticos y los recursos dinámicos no tienen relación con los recursos del ensamblado.

GetManifestResourceNames devuelve los nombres de todos los recursos en el ensamblado.

Archivos .resources

Los archivos .resources son contenedores de contenido potencialmente localizable. Un archivo .resources acaba siendo un recurso incrustado dentro de un ensamblado, al igual que cualquier otro tipo de archivo. La diferencia es que debes hacer lo siguiente:

- Empaquetar tu contenido en el archivo .resources desde el principio.
- Acceder a su contenido a través de un ResourceManager o una URI de paquete en lugar de utilizar GetManifestResourceStream.

Los archivos .resources tienen una estructura binaria y no son editables por humanos; por lo tanto, debes confiar en las herramientas proporcionadas por .NET y Visual Studio para trabajar con ellos.

El enfoque estándar con cadenas o tipos de datos simples es utilizar el formato .resx, que se puede convertir en un archivo .resources tanto con Visual Studio como con la herramienta resgen. El formato .resx también es adecuado para imágenes destinadas a una aplicación de Windows Forms o ASP.NET.

En una aplicación de WPF, debes utilizar la acción de compilación "Resource" de Visual Studio para imágenes u otro contenido similar que necesite ser referenciado mediante URI. Esto se aplica tanto si se necesita localización como si no.

Describiremos cómo hacer cada una de estas acciones en las siguientes secciones.

Archivos .resx

Un archivo .resx es un formato de diseño para producir archivos .resources en tiempo de diseño. Un archivo .resx utiliza XML y se estructura con pares de nombre/valor de la siguiente manera:

```
<root>
  <data name="Greeting">
    <value>hello</value>
  </data>
  <data name="DefaultFontSize" type="System.Int32, mscorlib">
    <value>10</value>
  </data>
</root>
```

Para crear un archivo .resx en Visual Studio, añade un elemento de proyecto de tipo "Resources File" (Archivo de recursos). El resto del trabajo se realiza automáticamente:

- Se crea la cabecera correcta.
- Se proporciona un diseñador para añadir cadenas, imágenes, archivos y otros tipos de datos.
- El archivo .resx se convierte automáticamente al formato .resources y se incrusta en el ensamblado al compilar.
- Se genera automáticamente una clase para ayudarte a acceder a los datos más adelante.



El diseñador de recursos añade imágenes como objetos de tipo Image (System.Drawing.dll) en lugar de matrices de bytes, lo que las hace inadecuadas para aplicaciones de WPF.

Lectura de archivos .resources



Si creas un archivo .resx en Visual Studio, se generará automáticamente una clase con el mismo nombre y propiedades para recuperar cada uno de sus elementos.

La clase ResourceManager lee archivos .resources incrustados en un ensamblado:

```
ResourceManager r = new ResourceManager ("welcome",
                                         Assembly.GetExecutingAssembly());
```

(El primer argumento debe tener el prefijo de espacio de nombres si el recurso se compiló en Visual Studio).

Luego puedes acceder a lo que hay dentro llamando a GetString o GetObject con una conversión de tipo:

```
string greeting = r.GetString ("Greeting");
int fontSize = (int) r.GetObject ("DefaultFontSize");
Image image = (Image) r.GetObject ("flag.png");
```

Para enumerar el contenido de un archivo .resources:

```
ResourceManager r = new ResourceManager (...);
ResourceSet set = r.GetResourceSet (CultureInfo.CurrentUICulture,
                                    true, true);
foreach (System.Collections.DictionaryEntry entry in set)
  Console.WriteLine (entry.Key);
```

Creación de un recurso con URI de paquete en Visual Studio

En una aplicación de WPF, los archivos XAML deben poder acceder a los recursos mediante URI. Por ejemplo:

```
<Button>
  <Image Height="50" Source="flag.png"/>
</Button>
```

O, si el recurso está en otra biblioteca:

```
<Button>
  <Image Height="50" Source="UtilsAssembly;Component/flag.png"/>
</Button>
```

(`Component` es una palabra clave literal).

Para crear recursos que se puedan cargar de esta manera, no puedes usar archivos .resx. En su lugar, debes agregar los archivos a tu proyecto y establecer su acción de compilación como "Resource" (no "Embedded Resource"). Visual Studio los compilará en un archivo .resources llamado `<NombreEnsamblado>.g.resources", que también es el hogar de los archivos XAML compilados (.baml).

Para cargar un recurso con clave URI de forma programática, llama a `Application.GetResourceStream`:

```
Uri u = new Uri ("flag.png", UriKind.Relative);
using (Stream s = Application.GetResourceStream (u).Stream)
```

Observa que se utilizó una URI relativa. También puedes utilizar una URI absoluta en el siguiente formato (las tres comas no son un error tipográfico):

```
Uri u = new Uri ("pack://application:,,,/flag.png");
```

Si prefieres especificar un objeto Assembly, puedes recuperar el contenido utilizando un ResourceManager:

```
Assembly a = Assembly.GetExecutingAssembly();
ResourceManager r = new ResourceManager (a.GetName().Name + ".g", a);
using (Stream s = r.GetStream ("flag.png"))
  ...
```

Un ResourceManager también te permite enumerar el contenido de un contenedor .g.resources dentro de un ensamblado específico.

Ensamblados satélite

Los datos incrustados en archivos .resources son localizables.

La localización de recursos es relevante cuando tu aplicación se ejecuta en una versión de Windows diseñada para mostrar todo en un idioma diferente. Para mantener la coherencia, tu aplicación también debería utilizar ese mismo idioma.

Una configuración típica es la siguiente:

- El ensamblado principal contiene archivos .resources para el idioma predeterminado o de respaldo.
- Los ensamblados satélite separados contienen archivos .resources localizados traducidos a diferentes idiomas.

Cuando tu aplicación se ejecuta, .NET examina el idioma del sistema operativo actual (a través de CultureInfo.CurrentUICulture). Cada vez que solicitas un recurso utilizando ResourceManager, el tiempo de ejecución busca un ensamblado satélite localizado. Si uno está disponible y contiene la clave del recurso solicitado, se utiliza en lugar de la versión del ensamblado principal.

Esto significa que puedes mejorar el soporte de idiomas simplemente agregando nuevos ensamblados satélite, sin necesidad de modificar el ensamblado principal.



Un ensamblado satélite no puede contener código ejecutable, solo recursos.

Los ensamblados satélite se implementan en subdirectorios de la carpeta del ensamblado, de la siguiente manera:

```
programBaseFolder\MyProgram.exe
                 \MyLibrary.exe
                 \XX\MyProgram.resources.dll
                 \XX\MyLibrary.resources.dll
```

XX se refiere al código de idioma de dos letras (como "de" para alemán) o a un código de idioma y región (como "en-GB" para inglés en Gran Bretaña). Este sistema de nomenclatura permite que el CLR encuentre y cargue automáticamente el ensamblado satélite correcto.

Generación de ensamblados satélite

Recuerda nuestro ejemplo anterior de .resx, que incluía lo siguiente:

```
<root>
  ...
  <data name="Greeting"
    <value>hello</value>
  </data>
</root>
```

Luego recuperamos el saludo en tiempo de ejecución de la siguiente manera:

```
ResourceManager r = new ResourceManager ("welcome",
                                         Assembly.GetExecutingAssembly());
Console.Write (r.GetString ("Greeting"));
```

Supongamos que en su lugar queremos escribir "hallo" si se ejecuta en la versión alemana de Windows. El primer paso es agregar otro archivo .resx llamado welcome.de.resx que sustituya "hello" por "hallo":

```
<root>
  <data name="Greeting">
    <value>hallo<value>
  </data>
</root>
```

En Visual Studio, esto es todo lo que necesitas hacer. Cuando reconstruyas el proyecto, se creará automáticamente un ensamblado satélite llamado MyApp.resources.dll en un subdirectorio llamado "de".

Pruebas de ensamblados satélite

Para simular la ejecución en un sistema operativo con un idioma diferente, debes cambiar CurrentUICulture utilizando la clase Thread:

```
System.Threading.Thread.CurrentThread.CurrentUICulture
  = new System.Globalization.CultureInfo ("de");
```

CultureInfo.CurrentUICulture es una versión de solo lectura de la misma propiedad.



Una estrategia de prueba útil es ℓo¢αℓïzɘ palabras que todavía se puedan leer en inglés pero que no utilicen los caracteres estándar del alfabeto romano.

Soporte del diseñador de Visual Studio

Los diseñadores en Visual Studio proporcionan un soporte extendido para la localización de componentes y elementos visuales. El diseñador de WPF tiene su propio flujo de trabajo para la localización; otros diseñadores basados en componentes utilizan una propiedad exclusiva del tiempo de diseño para hacer que parezca que un componente o control de Windows Forms tiene una propiedad Language. Para personalizarlo para otro idioma, simplemente cambia la propiedad Language y luego comienza a modificar el componente. Todas las propiedades de los controles que están etiquetados como Localizable se guardarán en un archivo .resx para ese idioma. Puedes cambiar entre idiomas en cualquier momento simplemente cambiando la propiedad Language.

Culturas y subculturas

Las culturas se dividen en culturas y subculturas. Una cultura representa un idioma en particular; una subcultura representa una variación regional de ese idioma. El tiempo de ejecución de .NET sigue el estándar RFC1766, que representa culturas y subculturas con códigos de dos letras. Aquí están los códigos para las culturas en inglés y alemán:

```
En
de
```

Aquí están los códigos para las subculturas del inglés australiano y el alemán austriaco:

```
en-AU
de-AT
```

Una cultura se representa en .NET con la clase System.Globalization.CultureInfo. Puedes examinar la cultura actual de tu aplicación de la siguiente manera:

```
Console.WriteLine (System.Threading.Thread.CurrentThread.CurrentCulture);
Console.WriteLine (System.Threading.Thread.CurrentThread.CurrentUICulture);
```

Al ejecutar esto en una computadora localizada para Australia, se ilustra la diferencia entre las dos:

```
en-AU
en-US
```

CurrentCulture refleja la configuración regional del Panel de control de Windows, mientras que CurrentUICulture refleja el idioma del sistema operativo.

La configuración regional incluye cosas como la zona horaria y el formato de la moneda y las fechas. CurrentCulture determina el comportamiento predeterminado de funciones como DateTime.Parse. La configuración regional se puede personalizar hasta el punto en que ya no se parezca a ninguna cultura en particular.

CurrentUICulture determina el idioma en el que la computadora se comunica con el usuario. Australia no necesita una versión separada del inglés para este propósito, por lo que simplemente utiliza la versión de Estados Unidos. Si pasara un par de meses trabajando en Austria, iría al Panel de control y cambiaría mi CurrentCulture a alemán austriaco. Sin embargo, dado que no hablo alemán, mi CurrentUICulture seguiría siendo inglés de Estados Unidos.

El ResourceManager, por defecto, utiliza la propiedad CurrentUICulture del subproceso actual para determinar la ensamblado satélite correcto que debe cargar. El ResourceManager utiliza un mecanismo de retroceso cuando carga recursos. Si se define un ensamblado de subcultura, se utiliza ese; de lo contrario, se retrocede a la cultura genérica. Si la cultura genérica no está presente, se retrocede a la cultura predeterminada en el ensamblado principal.

Carga, resolución y aislamiento de ensamblados

Cargar un ensamblado desde una ubicación conocida es un proceso relativamente sencillo. A esto lo llamamos carga de ensamblado.

Sin embargo, con mayor frecuencia, tú (o el CLR) deberán cargar un ensamblado solo conociendo su nombre completo (o simple). Esto se llama resolución de ensamblados. La resolución de ensamblados difiere de la carga en que primero se debe ubicar el ensamblado.

La resolución de ensamblados se activa en dos escenarios:

- Por el CLR, cuando necesita resolver una dependencia.
- Explícitamente, cuando llamas a un método como Assembly.Load(AssemblyName).

Para ilustrar el primer escenario, consideremos una aplicación que consta de un ensamblado principal y un conjunto de ensamblados de biblioteca referenciados estáticamente (dependencias), como se muestra en este ejemplo:

```
AdventureGame.dll    // Main assembly
Terrain.dll          // Referenced assembly
UIEngine.dll         // Referenced assembly
```

Cuando decimos "referenciados estáticamente", nos referimos a que AdventureGame.dll se compiló con referencias a Terrain.dll y UIEngine.dll. El compilador en sí no necesita realizar la resolución de ensamblados, porque se le indica (ya sea explícitamente o mediante MSBuild) dónde encontrar Terrain.dll y UIEngine.dll. Durante la compilación, escribe los nombres

completos de los ensamblados Terrain y UIEngine en los metadatos de AdventureGame.dll, pero no proporciona información sobre dónde encontrarlos. Por lo tanto, en tiempo de ejecución, los ensamblados Terrain y UIEngine deben resolverse.

La carga y resolución de ensamblados se manejan mediante un contexto de carga de ensamblados (ALC), específicamente una instancia de la clase AssemblyLoadContext en System.Runtime.Loader. Como AdventureGame.dll es el ensamblado principal de la aplicación, el CLR utiliza el ALC predeterminado (AssemblyLoadContext.Default) para resolver sus dependencias. El ALC predeterminado resuelve las dependencias primero buscando y examinando un archivo llamado AdventureGame.deps.json (que describe dónde encontrar las dependencias), o si no está presente, busca en la carpeta base de la aplicación, donde encontrará Terrain.dll y UIEngine.dll. (El ALC predeterminado también resuelve los ensamblados del tiempo de ejecución de .NET).

Como desarrollador, puedes cargar dinámicamente ensamblados adicionales durante la ejecución de tu programa. Por ejemplo, es posible que desees empaquetar características opcionales en ensamblados que solo se implementen cuando esas características se hayan comprado. En ese caso, podrías cargar los ensamblados adicionales, cuando estén presentes, llamando a Assembly.Load(AssemblyName).

Un ejemplo más complejo sería implementar un sistema de complementos mediante el cual el usuario puede proporcionar ensamblados de terceros que tu aplicación detecta y carga en tiempo de ejecución para ampliar la funcionalidad de tu aplicación. La complejidad surge porque cada ensamblado de complemento puede tener sus propias dependencias que también deben resolverse. Al subclasificar AssemblyLoadContext y anular su método de resolución de ensamblados (Load), puedes controlar cómo un complemento encuentra sus dependencias. Por ejemplo, podrías decidir que cada complemento debe residir en su propia carpeta, y sus dependencias también deben residir en esa carpeta.

Los ALC también tienen otro propósito: al instanciar un AssemblyLoadContext separado para cada (complemento + dependencias), puedes mantenerlos aislados, asegurando que sus dependencias se carguen en paralelo y no interfieran entre sí (ni con la aplicación principal). Cada uno, por ejemplo, puede tener su propia versión de JSON.NET. Por lo tanto, además de la carga y la resolución, los ALC también proporcionan un mecanismo de aislamiento.

En ciertas condiciones, los ALC incluso se pueden descargar, liberando su memoria.

En esta sección, elaboramos sobre cada uno de estos principios y describimos lo siguiente:

- Cómo los ALC manejan la carga y resolución de ensamblados.
- El papel del ALC predeterminado.
- Assembly.Load y ALCs contextuales.
- Cómo utilizar AssemblyDependencyResolver.
- Cómo cargar y resolver bibliotecas sin administración.
- Descarga de ALCs.
- Los métodos de carga de ensamblados heredados.

Luego, ponemos en práctica la teoría y demostramos cómo escribir un sistema de complementos con aislamiento de ALC.



La clase AssemblyLoadContext es nueva en .NET 5 y .NET Core. En .NET Framework, los ALC estaban presentes pero restringidos y ocultos: la única forma de crear e interactuar con ellos era indirectamente a través de los métodos estáticos LoadFile(string), LoadFrom(string) y Load(byte[]) en la clase Assembly. En comparación con la API de ALC, estos métodos son inflexibles y su uso puede dar lugar a sorpresas (especialmente al manejar dependencias). Por esta razón, es mejor favorecer el uso explícito de la API de AssemblyLoadContext en .NET 5 y .NET Core.

Contextos de carga de ensamblados

Como acabamos de discutir, la clase AssemblyLoadContext es responsable de cargar y resolver ensamblados, así como de proporcionar un mecanismo de aislamiento.

Cada objeto Assembly de .NET pertenece exactamente a un AssemblyLoadContext. Puedes obtener el ALC de un ensamblado de la siguiente manera:

```
Assembly assem = Assembly.GetExecutingAssembly();
AssemblyLoadContext context = AssemblyLoadContext.GetLoadContext (assem);
Console.WriteLine (context.Name);
```

De manera inversa, puedes pensar en un ALC como "contenedor" o "propietario" de ensamblados, que puedes obtener a través de su propiedad Assemblies. Siguiendo el ejemplo anterior:

```
foreach (Assembly a in context.Assemblies)
  Console.WriteLine (a.FullName);
```

La clase AssemblyLoadContext también tiene una propiedad estática All que enumera todos los ALCs.

Puedes crear un nuevo ALC simplemente instanciando AssemblyLoadContext y proporcionando un nombre (el nombre es útil para depurar). Sin embargo, lo más común es crear una subclase de AssemblyLoadContext para poder implementar la lógica de resolución de dependencias, es decir, cargar un ensamblado a partir de su nombre.

Cargando ensamblados

AssemblyLoadContext proporciona los siguientes métodos para cargar explícitamente un ensamblado en su contexto:

```
public Assembly LoadFromAssemblyPath (string assemblyPath);
public Assembly LoadFromStream (Stream assembly, Stream assemblySymbols);
```

El primer método carga un ensamblado desde una ruta de archivo, mientras que el segundo método lo carga desde un flujo (que puede provenir directamente de la memoria). El segundo parámetro es opcional y corresponde al contenido del archivo de depuración (.pdb) del proyecto,

que permite que las trazas de pila incluyan información del código fuente cuando se ejecuta el código (útil en la generación de informes de excepciones).

Con ambos métodos, no se realiza ninguna resolución.

El siguiente código carga el ensamblado c:\temp\foo.dll en su propio ALC:

```
var alc = new AssemblyLoadContext ("Test");
Assembly assem = alc.LoadFromAssemblyPath (@"c:\temp\foo.dll");
```

Si el ensamblado es válido, la carga siempre tendrá éxito, sujeto a una regla importante: el nombre simple de un ensamblado debe ser único dentro de su ALC. Esto significa que no puedes cargar varias versiones del mismo ensamblado con el mismo nombre en un solo ALC; para hacer esto, debes crear ALCs adicionales. Podríamos cargar otra copia de foo.dll de la siguiente manera:

```
var alc2 = new AssemblyLoadContext ("Test 2");
Assembly assem2 = alc2.LoadFromAssemblyPath (@"c:\temp\foo.dll");
```

Ten en cuenta que los tipos que provienen de diferentes objetos Assembly son incompatibles incluso si los ensamblados son idénticos en otros aspectos. En nuestro ejemplo, los tipos en assem son incompatibles con los tipos en assem2.

Después de cargar un ensamblado, no se puede descargar a menos que se descargue su ALC (ver "Descarga de ALCs" en la página 773). El CLR mantiene un bloqueo del archivo durante el tiempo que está cargado.



Puedes evitar bloquear el archivo cargando el ensamblado a través de una matriz de bytes:

```
bytes[] bytes = File.ReadAllBytes (@"c:\temp\foo.dll");
var ms = new MemoryStream (bytes);
var assem = alc.LoadFromStream (ms);
```

Esto tiene dos inconvenientes:

- La propiedad Location del ensamblado quedará en blanco. A veces es útil saber desde dónde se cargó un ensamblado (y algunas API dependen de que esté poblada).
- El consumo de memoria privada debe aumentar inmediatamente para acomodar el tamaño completo del ensamblado. Si en su lugar cargas desde un nombre de archivo, el CLR utiliza un archivo mapeado en memoria, lo que permite la carga diferida y el intercambio de procesos. Además, si la memoria escasea, el sistema operativo puede liberar su memoria y volver a cargarla según sea necesario sin escribir en un archivo de paginación.

LoadFromAssemblyName

AssemblyLoadContext también proporciona el siguiente método, que carga un ensamblado por su nombre:

```
public Assembly LoadFromAssemblyName (AssemblyName assemblyName);
```

A diferencia de los dos métodos anteriores, no proporcionas ninguna información para indicar dónde se encuentra el ensamblado; en su lugar, estás instruyendo al ALC que resuelva el ensamblado.

Resolución de ensamblados

El método anterior desencadena la resolución de ensamblados. El CLR también desencadena la resolución de ensamblados al cargar dependencias. Por ejemplo, supongamos que el ensamblado A hace referencia estáticamente al ensamblado B. Para resolver la referencia a B, el CLR desencadena la resolución de ensamblados en el ALC en el que se cargó el ensamblado A.


El CLR resuelve las dependencias desencadenando la resolución de ensamblados, ya sea que el ensamblado desencadenante se encuentre en el ALC predeterminado o en un ALC personalizado. La diferencia radica en que con el ALC predeterminado, las reglas de resolución están codificadas de forma fija, mientras que con un ALC personalizado, tú mismo escribes las reglas.

Aquí está lo que sucede:

1. El CLR primero verifica si ya se ha producido una resolución idéntica en ese ALC (con un nombre completo de ensamblado coincidente); si es así, devuelve el ensamblado que se devolvió anteriormente.

2. De lo contrario, llama al método Load del ALC (virtual protegido), que se encarga de localizar y cargar el ensamblado. El método Load del ALC predeterminado aplica las reglas que describimos en "El ALC predeterminado". Con un ALC personalizado, depende completamente de ti cómo localizar el ensamblado. Por ejemplo, puedes buscar en alguna carpeta y luego llamar a LoadFromAssemblyPath cuando encuentres el ensamblado. También es perfectamente válido devolver un ensamblado ya cargado desde el mismo ALC o desde otro (lo demostramos en "Escribiendo un sistema de complementos").

3. Si el paso 2 devuelve null, el CLR luego llama al método Load en el ALC predeterminado (esto sirve como una útil "alternativa" para resolver los ensamblados de tiempo de ejecución de .NET y los ensamblados comunes de la aplicación).

4. Si el paso 3 devuelve null, el CLR dispara los eventos de resolución (Resolving) en ambos ALC, primero en el ALC predeterminado y luego en el ALC original.

5. (Para compatibilidad con .NET Framework): Si el ensamblado aún no se ha resuelto, se dispara el evento AppDomain.CurrentDomain.AssemblyResolve.


Después de que se complete este proceso, el CLR realiza una "comprobación de coherencia" para asegurarse de que el ensamblado cargado tenga un nombre compatible con el

solicitado. El nombre simple debe coincidir; el token de clave pública debe coincidir si se especifica. La versión no necesita coincidir, puede ser más alta o más baja que la solicitada.

A partir de esto, podemos ver que hay dos formas de implementar la resolución de ensamblados en un ALC personalizado:

- Sobrescribir el método Load del ALC. Esto le da a tu ALC la oportunidad de tener "la última palabra" sobre lo que sucede, lo cual es generalmente deseable (y esencial cuando se necesita aislamiento).
- Manejar el evento Resolving del ALC. Esto se dispara solo después de que el ALC predeterminado no pudo resolver el ensamblado.



Si adjuntas múltiples controladores de eventos al evento Resolving, el primero que devuelva un valor no nulo será el ganador.

Para ilustrar, supongamos que queremos cargar un ensamblado que nuestra aplicación principal no conoce en tiempo de compilación, llamado foo.dll, ubicado en c:\temp (que es diferente a la carpeta de nuestra aplicación). También supongamos que foo.dll tiene una dependencia privada en bar.dll. Queremos asegurarnos de que cuando carguemos c:\temp\foo.dll y ejecutemos su código, c:\temp\bar.dll se resuelva correctamente. También queremos asegurarnos de que foo y su dependencia privada, bar, no interfieran con la aplicación principal.

Comencemos escribiendo un ALC personalizado que sobrescribe el método Load:

```
using System.IO;
using System.Runtime.Loader;

class FolderBasedALC : AssemblyLoadContext


{
  readonly string _folder;
  public FolderBasedALC (string folder) => _folder = folder;

  protected override Assembly Load (AssemblyName assemblyName)
  {
    // Attempt to find the assembly:
    string targetPath = Path.Combine (_folder, assemblyName.Name + ".dll");

    if (File.Exists (targetPath))
      return LoadFromAssemblyPath (targetPath);   // Load the assembly

    return null;    // We can't find it: it could be a .NET runtime assembly
  }
}
```

Es importante tener en cuenta que en el método Load, devolvemos null si el archivo del ensamblado no está presente. Esta verificación es importante porque foo.dll también tendrá dependencias en los ensamblados de .NET BCL (Biblioteca de Clases Base); por lo tanto, se llamará al método Load en ensamblados como System.Runtime. Al devolver null, permitimos que el CLR recurra al ALC predeterminado, que resolverá correctamente estos ensamblados.



Es importante destacar que no intentamos cargar los ensamblados de .NET BCL en nuestro propio ALC. Estos ensamblados del sistema no están diseñados para ejecutarse fuera del ALC predeterminado, y los intentos de cargarlos en tu propio ALC pueden provocar un comportamiento incorrecto, degradación del rendimiento e incompatibilidad inesperada de tipos.

Así es cómo podríamos usar nuestro ALC personalizado para cargar el ensamblado foo.dll en c:\temp:

```
var alc = new FolderBasedALC (@"c:\temp");
Assembly foo = alc.LoadFromAssemblyPath (@"c:\temp\foo.dll");
...
```

Cuando posteriormente comencemos a llamar código en el ensamblado foo, en algún momento el CLR necesitará resolver la dependencia en bar.dll. En ese momento, el método Load del ALC personalizado se activará y encontrará correctamente el ensamblado bar.dll en c:\temp.

En este caso, nuestro método Load también es capaz de resolver foo.dll, por lo que podríamos simplificar nuestro código a esto:

```
var alc = new FolderBasedALC (@"c:\temp");
Assembly foo = alc.LoadFromAssemblyName (new AssemblyName ("foo"));
...
```

Ahora, consideremos una solución alternativa: en lugar de heredar de AssemblyLoadContext y anular Load, podríamos instanciar un AssemblyLoadContext simple y manejar su evento Resolving:

```
var alc = new AssemblyLoadContext ("test");
alc.Resolving += (loadContext, assemblyName) =>
{
  string targetPath = Path.Combine (@"c:\temp", assemblyName.Name + ".dll");

  return alc.LoadFromAssemblyPath (targetPath);   // Load the assembly
};
Assembly foo = alc.LoadFromAssemblyName (new AssemblyName ("foo"));
```

Ahora notamos que no es necesario verificar si el ensamblado existe. Debido a que el evento Resolving se dispara después de que el ALC predeterminado ha tenido la oportunidad de resolver el ensamblado (y solo cuando falla), nuestro controlador no se ejecutará para los ensamblados de .NET BCL. Esto hace que esta solución sea más simple, aunque tiene una desventaja. Recuerda que en nuestro escenario, la aplicación principal no conocía nada acerca de foo.dll o bar.dll en tiempo de compilación. Esto significa que es posible que la aplicación principal dependa de ensamblados llamados foo.dll o bar.dll. Si esto ocurriera, el evento Resolving nunca se dispararía

y los ensamblados foo y bar de la aplicación se cargarían en su lugar. En otras palabras, no lograríamos el aislamiento deseado.



Nuestra clase FolderBasedALC es útil para ilustrar el concepto de resolución de ensamblados, pero tiene un uso limitado en la vida real porque no puede manejar dependencias específicas de la plataforma y dependencias de NuGet en tiempo de desarrollo para proyectos de bibliotecas. En la sección "AssemblyDependencyResolver" en la página 772, describimos la solución a este problema, y en "Writing a Plug-In System" en la página 775, ofrecemos un ejemplo detallado.

El ALC predeterminado

Cuando una aplicación se inicia, el CLR asigna un ALC especial a la propiedad estática AssemblyLoadContext.Default. El ALC predeterminado es donde se cargan el ensamblado de inicio y sus dependencias referenciadas estáticamente, así como los ensamblados de .NET runtime BCL.

El ALC predeterminado busca primero en las rutas de exploración predeterminadas para resolver automáticamente los ensamblados (consulta "Exploración predeterminada" en la página 767); esto generalmente corresponde a las ubicaciones indicadas en los archivos .deps.json y .runtimeconfig.json de la aplicación.

Si el ALC no puede encontrar un ensamblado en sus rutas de exploración predeterminadas, se dispara su evento Resolving. Al manejar este evento, puedes cargar el ensamblado desde otras ubicaciones, lo que significa que puedes implementar las dependencias de una aplicación en ubicaciones adicionales, como subcarpetas, carpetas compartidas o incluso como un recurso binario dentro del ensamblado principal:

```
AssemblyLoadContext.Default.Resolving += (loadContext, assemblyName) =>
{
  // Try to locate assemblyName, returning an Assembly object or null.
  // Typically you'd call LoadFromAssemblyPath after finding the file.
  // ...
};
```

El evento Resolving en el ALC predeterminado también se dispara cuando un ALC personalizado no puede resolver (es decir, cuando su método Load devuelve null) y el ALC predeterminado no puede resolver el ensamblado.

También puedes cargar ensamblados en el ALC predeterminado desde fuera del evento Resolving. Sin embargo, antes de continuar, debes determinar si puedes resolver el problema de manera más efectiva utilizando un ALC separado o los enfoques que describimos en la siguiente sección (que utilizan los ALC de ejecución y contextuales). Codificar directamente en el ALC predeterminado hace que tu código sea frágil, ya que no se puede aislar por completo (por ejemplo, mediante frameworks de pruebas unitarias o LINQPad).

Si aún deseas continuar, es preferible llamar a un método de resolución (por ejemplo, LoadFromAssemblyName) en lugar de un método de carga (como LoadFromAssemblyPath),

especialmente si el ensamblado se referencia estáticamente. Esto se debe a que es posible que el ensamblado ya esté cargado, en cuyo caso LoadFromAssemblyName devolverá el ensamblado ya cargado, mientras que LoadFromAssemblyPath lanzará una excepción.

(Con LoadFromAssemblyPath, también corres el riesgo de cargar el ensamblado desde un lugar que no sea coherente con el lugar donde el mecanismo de resolución predeterminado del ALC lo encontraría).

Si el ensamblado se encuentra en un lugar donde el ALC no lo encontrará automáticamente, aún puedes seguir este procedimiento y además manejar el evento Resolving del ALC.

Ten en cuenta que al llamar a LoadFromAssemblyName, no es necesario proporcionar el nombre completo; el nombre simple es suficiente (y es válido incluso si el ensamblado tiene un nombre fuerte). Por ejemplo:

```
AssemblyLoadContext.Default.LoadFromAssemblyName ("System.Xml");
```

Sin embargo, si incluyes el token de clave pública en el nombre, debe coincidir con lo que está cargado.

Exploración predeterminada (Default probing)

Las rutas de exploración predeterminadas normalmente incluyen lo siguiente:

- Las rutas especificadas en AppName.deps.json (donde AppName es el nombre del ensamblado principal de tu aplicación). Si este archivo no está presente, se utiliza la carpeta base de la aplicación.
- Las carpetas que contienen los ensamblados del sistema de .NET runtime (si tu aplicación depende del Framework).

MSBuild genera automáticamente un archivo llamado AppName.deps.json, que describe dónde encontrar todas las dependencias de la aplicación. Estas incluyen ensamblados independientes de la plataforma, que se colocan en la carpeta base de la aplicación, y ensamblados específicos de la plataforma, que se colocan en el subdirectorio runtimes\ en una subcarpeta como win o unix.

Las rutas especificadas en el archivo .deps.json generado son relativas a la carpeta base de la aplicación, o a cualquier carpeta adicional que especifiques en la sección additionalProbingPaths de los archivos de configuración AppName.runtimeconfig.json y/o AppName.runtimeconfig.dev.json (este último está destinado solo al entorno de desarrollo).

El ALC "Actual"

En la sección anterior, advertimos contra la carga explícita de ensamblados en el ALC predeterminado. Lo que normalmente deseas hacer en su lugar es cargar/resolver en el ALC "actual".

En la mayoría de los casos, el ALC "actual" es aquel que contiene el ensamblado que se está ejecutando actualmente:

```
var executingAssem = Assembly.GetExecutingAssembly();
var alc = AssemblyLoadContext.GetLoadContext (executingAssem);

Assembly assem = alc.LoadFromAssemblyName (...);  // to resolve by name
        // OR: = alc.LoadFromAssemblyPath (...);  // to load by path
```

Aquí tienes una forma más flexible y explícita de obtener el ALC:

```
var myAssem = typeof (SomeTypeInMyAssembly).Assembly;
var alc = AssemblyLoadContext.GetLoadContext (myAssem);
...
```

A veces es imposible inferir el ALC "actual". Por ejemplo, supongamos que eres responsable de escribir el serializador binario de .NET (describimos la serialización en el suplemento en línea en http://www.albahari.com/nutshell). Un serializador como este escribe los nombres completos de los tipos que serializa (incluidos los nombres de sus ensamblados), que deben resolverse durante la deserialización. La pregunta es, ¿qué ALC deberías usar? El problema de depender del ensamblado en ejecución es que devolverá el ensamblado que contiene el deserializador, no el ensamblado que llama al deserializador.

La mejor solución no es adivinar, sino preguntar:

```
public object Deserialize (Stream stream, AssemblyLoadContext alc)
{
  ...
}
```

Ser explícito maximiza la flexibilidad y minimiza la posibilidad de cometer errores. El llamante ahora puede decidir qué debe considerarse el ALC "actual":

```
var assem = typeof (SomeTypeThatIWillBeDeserializing).Assembly;
var alc = AssemblyLoadContext.GetLoadContext (assem);
var object = Deserialize (someStream, alc);
```

Carga de ensamblados y ALCs contextuales

Para ayudar con el caso común de cargar un ensamblado en el ALC que se está ejecutando actualmente:

```
var executingAssem = Assembly.GetExecutingAssembly();
var alc = AssemblyLoadContext.GetLoadContext (executingAssem);
Assembly assem = alc.LoadFromAssemblyName (...);
```

Microsoft ha definido el siguiente método en la clase Assembly:

```
public static Assembly Load (string assemblyString);
```

También hay una versión funcionalmente idéntica que acepta un objeto AssemblyName:

```
public static Assembly Load (AssemblyName assemblyRef);
```

(No confundas estos métodos con el método Load(byte[]) heredado, que se comporta de manera totalmente diferente, consulta "Los métodos de carga heredados" en la página 774).

Al igual que con LoadFromAssemblyName, tienes la opción de especificar el nombre simple, parcial o completo del ensamblado:

```
Assembly a = Assembly.Load ("System.Private.Xml");
```

Esto carga el ensamblado System.Private.Xml en cualquier ALC en el que esté cargado el ensamblado del código en ejecución.

En este caso, especificamos un nombre simple. Las siguientes cadenas también serían válidas y tendrían el mismo resultado en .NET:

```
"System.Private.Xml, PublicKeyToken=cc7b13ffcd2ddd51"
"System.Private.Xml, Version=4.0.1.0"
"System.Private.Xml, Version=4.0.1.0, PublicKeyToken=cc7b13ffcd2ddd51"
```

Si eliges especificar un token de clave pública, debe coincidir con lo que está cargado.



La documentación de Microsoft Developer Network (MSDN) advierte que no se debe cargar un ensamblado a partir de un nombre parcial y recomienda especificar la versión exacta y el token de clave pública. Su razonamiento se basa en factores relevantes para .NET Framework, como los efectos del Global Assembly Cache y la seguridad de acceso al código. En .NET 5 y .NET Core, estos factores no están presentes y generalmente es seguro cargar desde un nombre simple o parcial.

Ambos métodos son estrictamente para resolución, por lo que no puedes especificar una ruta de archivo (si completas la propiedad CodeBase en el objeto AssemblyName, se ignorará).



No caigas en la trampa de usar Assembly.Load para cargar un ensamblado referenciado estáticamente. Todo lo que necesitas hacer en este caso es hacer referencia a un tipo en el ensamblado y obtener el ensamblado a partir de eso:

```
Assembly a = typeof (System.Xml.Formatting).Assembly;
```

O incluso puedes hacer esto:

```
Assembly a = System.Xml.Formatting.Indented.GetType().Assembly;
```

Esto evita codificar el nombre del ensamblado (que podrías cambiar en el futuro) al tiempo que activa la resolución del ensamblado en el ALC del código en ejecución (como sucedería con Assembly.Load).

Si tuvieras que escribir el método Assembly.Load tú mismo, se vería (casi) como esto:

```
[MethodImpl(MethodImplOptions.NoInlining)]
Assembly Load (string name)
```

```
{
  Assembly callingAssembly = Assembly.GetCallingAssembly();
  var callingAlc = AssemblyLoadContext.GetLoadContext (callingAssembly);
  return callingAlc.LoadFromAssemblyName (new AssemblyName (name));
}
```

EnterContextualReflection

El enfoque de Assembly.Load de utilizar el contexto de carga de la ALC del ensamblado llamador falla cuando se llama a Assembly.Load a través de un intermediario, como un deserializador o un ejecutor de pruebas unitarias. Si el intermediario está definido en un ensamblado diferente, se utiliza el contexto de carga del intermediario en lugar del contexto de carga del llamador.



Describimos este escenario anteriormente cuando hablamos sobre cómo podrías escribir un deserializador. En tales casos, la solución ideal es obligar al llamador a especificar un contexto de carga de ALC en lugar de inferirlo con Assembly.Load(string).

Sin embargo, debido a que .NET 5 y .NET Core evolucionaron a partir de .NET Framework, donde el aislamiento se lograba con dominios de aplicación en lugar de ALC, la solución ideal no es prevalente y Assembly.Load(string) se utiliza a veces de manera inapropiada en escenarios en los que no se puede inferir de manera confiable el contexto de carga de ALC. Un ejemplo de esto es el serializador binario de .NET.

Para permitir que Assembly.Load siga funcionando en tales escenarios, Microsoft ha agregado un método a AssemblyLoadContext llamado EnterContextualReflection. Esto asigna un contexto de carga de ALC a AssemblyLoadContext.CurrentContextualReflectionContext.

Aunque esta es una propiedad estática, su valor se almacena en una variable AsyncLocal, por lo que puede contener valores separados en diferentes hilos (pero aún se conserva en operaciones asíncronas).

Si esta propiedad no es nula, Assembly.Load la utiliza automáticamente en lugar del contexto de carga del llamador:

```
Method1();

var myALC = new AssemblyLoadContext ("test");
using (myALC.EnterContextualReflection())
{
   Console.WriteLine (
     AssemblyLoadContext.CurrentContextualReflectionContext.Name);  // test

   Method2();
}

// Once disposed, EnterContextualReflection() no longer has an effect.
Method3();

void Method1() => Assembly.Load ("...");    // Will use calling ALC
void Method2() => Assembly.Load ("...");    // Will use myALC
void Method3() => Assembly.Load ("...");    // Will use calling ALC
```

Previamente demostramos cómo podrías escribir un método que es funcionalmente similar a Assembly.Load. Aquí tienes una versión más precisa que tiene en cuenta el contexto de reflexión contextual:

```
[MethodImpl(MethodImplOptions.NoInlining)]
Assembly Load (string name)
{
  var alc = AssemblyLoadContext.CurrentContextualReflectionContext
     ?? AssemblyLoadContext.GetLoadContext (Assembly.GetCallingAssembly());

  return alc.LoadFromAssemblyName (new AssemblyName (name));
}
```

Aunque el contexto de reflexión contextual puede ser útil para permitir que se ejecute código heredado, una solución más robusta (como describimos anteriormente) es modificar el código que llama a Assembly.Load para que en su lugar llame a LoadFromAssemblyName en un ALC que sea proporcionado por el llamador.



.NET Framework no tiene un equivalente de EnterContextualReflection, y no lo necesita, a pesar de tener los mismos métodos Assembly.Load. Esto se debe a que con .NET Framework, el aislamiento se logra principalmente con dominios de aplicación en lugar de ALC. Los dominios de aplicación proporcionan un modelo de aislamiento más sólido en el que cada dominio de aplicación tiene su propio contexto de carga predeterminado, por lo que el aislamiento puede funcionar incluso cuando solo se utiliza el contexto de carga predeterminado.

Carga y resolución de bibliotecas no administradas

Las ALC también pueden cargar y resolver bibliotecas nativas. La resolución nativa se activa cuando llamas a un método externo que está marcado con el atributo [DllImport]:

```
[DllImport ("SomeNativeLibrary.dll")]
static extern int SomeNativeMethod (string text);
```

Como no especificamos una ruta completa en el atributo [DllImport], llamar a SomeNativeMethod desencadena una resolución en cualquier ALC que contenga el ensamblado en el que se define SomeNativeMethod.

El método virtual de resolución en la ALC se llama LoadUnmanagedDll, y el método de carga se llama LoadUnmanagedDllFromPath:

```
protected override IntPtr LoadUnmanagedDll (string unmanagedDllName)
{
  // Locate the full path of unmanagedDllName...
  string fullPath = ...
  return LoadUnmanagedDllFromPath (fullPath);    // Load the DLL
}
```

Si no puedes encontrar el archivo, puedes devolver IntPtr.Zero. El CLR entonces disparará el evento ResolvingUnmanagedDll de la ALC.

Curiosamente, el método LoadUnmanagedDllFromPath es protegido, por lo que normalmente no podrás llamarlo desde un controlador de eventos ResolvingUnmanagedDll. Sin embargo, puedes lograr el mismo resultado llamando al método estático NativeLibrary.Load:

```
someALC.ResolvingUnmanagedDll += (requestingAssembly, unmanagedDllName) =>
{
  return NativeLibrary.Load ("(full path to unmanaged DLL)");
};
```

Aunque las bibliotecas nativas generalmente se resuelven y cargan mediante ALC, no "pertenecen" a un ALC. Después de cargarse, una biblioteca nativa se mantiene por sí sola y asume la responsabilidad de resolver cualquier dependencia transitoria que pueda tener. Además, las bibliotecas nativas son globales para el proceso, por lo que no es posible cargar dos versiones diferentes de una biblioteca nativa si tienen el mismo nombre de archivo.

AssemblyDependencyResolver

En la sección "Default probing" de la página 767, mencionamos que el ALC predeterminado lee los archivos .deps.json y .runtimeconfig.json, si están presentes, para determinar dónde buscar y resolver dependencias de NuGet específicas de la plataforma y del tiempo de desarrollo.

Si deseas cargar un ensamblado en un ALC personalizado que tiene dependencias específicas de la plataforma o de NuGet, deberás reproducir de alguna manera esta lógica. Podrías lograrlo analizando los archivos de configuración y siguiendo cuidadosamente las reglas de los identificadores específicos de la plataforma, pero hacerlo no solo es difícil, sino que el código que escribas se romperá si las reglas cambian en una versión posterior de .NET.

La clase AssemblyDependencyResolver resuelve este problema. Para utilizarla, la instancias con la ruta del ensamblado cuyas dependencias deseas analizar:

```
var resolver = new AssemblyDependencyResolver (@"c:\temp\foo.dll");
```

Luego, para encontrar la ruta de una dependencia, llamas al método ResolveAssemblyToPath:

```
string path = resolver.ResolveAssemblyToPath (new AssemblyName ("bar"));
```

En ausencia de un archivo .deps.json (o si el .deps.json no contiene nada relevante para bar.dll), esto dará como resultado c:\temp\bar.dll.

De manera similar, puedes resolver dependencias no administradas llamando a ResolveUnmanagedDllToPath.

Una excelente manera de ilustrar un escenario más complejo es crear un nuevo proyecto de Consola llamado ClientApp y luego agregar una referencia de NuGet a Microsoft.Data.SqlClient. Agrega la siguiente clase:

```
using Microsoft.Data.SqlClient;

namespace ClientApp
{
  public class Program
  {
    public static SqlConnection GetConnection() => new SqlConnection();

    static void Main() => GetConnection();   // Test that it resolves
  }
}
```

Ahora compila la aplicación y busca en la carpeta de salida: verás un archivo llamado Microsoft.Data.SqlClient.dll. Sin embargo, este archivo nunca se carga cuando se ejecuta y cualquier intento de cargarlo explícitamente arrojará una excepción. El ensamblado que se carga realmente se encuentra en la subcarpeta runtimes\win (o runtimes/unix); el ALC predeterminado sabe cómo cargarlo porque analiza el archivo ClientApp.deps.json.

Si intentaras cargar el ensamblado ClientApp.dll desde otra aplicación, necesitarías escribir un ALC que pueda resolver su dependencia, Microsoft.Data.SqlClient.dll. Para hacerlo, sería insuficiente simplemente buscar en la carpeta donde se encuentra ClientApp.dll (como hicimos en "Resolviendo ensamblados" en la página 763). En su lugar, deberías usar AssemblyDependencyResolver para determinar dónde se encuentra ese archivo para la plataforma en uso:

```
string path = @"C:\source\ClientApp\bin\Debug\netcoreapp3.0\ClientApp.dll";
var resolver = new AssemblyDependencyResolver (path);
var sqlClient = new AssemblyName ("Microsoft.Data.SqlClient");
Console.WriteLine (resolver.ResolveAssemblyToPath (sqlClient));
```

En una máquina con Windows, esto producirá la siguiente salida:

```
C:\source\ClientApp\bin\Debug\netcoreapp3.0\runtimes\win\lib\netcoreapp2.1
\Microsoft.Data.SqlClient.dll
```

Proporcionamos un ejemplo completo en "Escribir un sistema de complementos" en la página 775.

Descarga de ALCs (Unloading ALCs)

En casos simples, es posible descargar un AssemblyLoadContext que no sea el predeterminado, liberando memoria y liberando los bloqueos de archivos de los ensamblados que cargó. Para que esto funcione, el ALC debe haberse instanciado con el parámetro isCollectible en true:

```
var alc = new AssemblyLoadContext ("test", isCollectible:true);
```

Luego puedes llamar al método Unload en el ALC para iniciar el proceso de descarga.

El modelo de descarga es cooperativo en lugar de preemptivo. Si hay métodos ejecutándose en cualquiera de los ensamblados del ALC, la descarga se pospondrá hasta que esos métodos finalicen.

La descarga real se lleva a cabo durante la recolección de basura; no se llevará a cabo si cualquier cosa fuera del ALC tiene alguna referencia (no débil) a cualquier cosa dentro del ALC (incluyendo objetos, tipos y ensamblados). No es raro que las API (incluidas las del BCL de .NET) almacenen en caché objetos en campos estáticos o diccionarios, o se suscriban a eventos, lo que facilita crear referencias que eviten una descarga, especialmente si el código en el ALC utiliza API fuera de su ALC de manera compleja. Determinar la causa de una descarga fallida es difícil y requiere el uso de herramientas como WinDbg.

Los métodos de carga heredados (The Legacy Loading Methods)

Si aún estás utilizando .NET Framework (o estás escribiendo una biblioteca que apunta a .NET Standard y deseas admitir .NET Framework), no podrás utilizar la clase AssemblyLoadContext. En su lugar, la carga se realiza utilizando los siguientes métodos:

```
public static Assembly LoadFrom (string assemblyFile);
public static Assembly LoadFile (string path);
public static Assembly Load (byte[] rawAssembly);
```

LoadFile y Load(byte[]) proporcionan aislamiento, mientras que LoadFrom no lo hace. La resolución se realiza mediante el manejo del evento AssemblyResolve del dominio de aplicación, que funciona de manera similar al evento Resolving del ALC predeterminado.

El método Assembly.Load(string) también está disponible para desencadenar la resolución y funciona de manera similar.

LoadFrom

LoadFrom carga un ensamblado desde una ruta dada en el ALC predeterminado. Es similar a llamar a AssemblyLoadContext.Default.LoadFromAssemblyPath, excepto por lo siguiente:

- Si ya hay un ensamblado con el mismo nombre simple presente en el ALC predeterminado, LoadFrom devuelve ese ensamblado en lugar de arrojar una excepción.
- Si no hay un ensamblado con el mismo nombre simple presente en el ALC predeterminado y se realiza una carga, al ensamblado se le asigna un estado especial de "LoadFrom". Este estado afecta la lógica de resolución del ALC predeterminado, de manera que si ese ensamblado tiene dependencias en la misma carpeta, esas dependencias se resolverán automáticamente.


.NET Framework tiene un Global Assembly Cache (GAC). Si el ensamblado está presente en el GAC, el CLR siempre lo cargará desde allí en su lugar. Esto se aplica a los tres métodos de carga.

La capacidad de LoadFrom para resolver automáticamente las dependencias transitivas en la misma carpeta puede ser conveniente, hasta que carga un ensamblado que no debería. Debido a que estos escenarios pueden ser difíciles de depurar, puede ser mejor utilizar Load(string) o LoadFile y resolver las dependencias transitivas mediante el manejo del evento AssemblyResolve del dominio de aplicación. Esto te brinda el poder de decidir cómo resolver cada ensamblado y permite la depuración (mediante la creación de un punto de interrupción dentro del controlador de eventos).

LoadFile y Load(byte[])

LoadFile y Load(byte[]) cargan un ensamblado desde una ruta de archivo o una matriz de bytes en un nuevo ALC. A diferencia de LoadFrom, estos métodos proporcionan aislamiento y te permiten cargar múltiples versiones del mismo ensamblado. Sin embargo, hay dos advertencias:

- Llamar a LoadFile nuevamente con la misma ruta devolverá la assembly cargada previamente.
- En .NET Framework, ambos métodos primero verifican el GAC y cargan desde allí si la assembly está presente.

Con LoadFile y Load(byte[]), obtienes un ALC separado por cada assembly (aunque hay excepciones). Esto permite el aislamiento, aunque puede complicar su gestión.

Para resolver dependencias, debes manejar el evento Resolving del dominio de la aplicación, el cual se activa en todos los ALCs:

```
AppDomain.CurrentDomain.AssemblyResolve += (sender, args) =>
{
  string fullAssemblyName = args.Name;
  // return an Assembly object or null
  ...
};
```

La variable args también incluye una propiedad llamada RequestingAssembly, que te indica qué assembly activó la resolución.

Después de localizar la assembly, puedes llamar a Assembly.LoadFile para cargarla.


Puedes enumerar todas las assemblies cargadas en el dominio de la aplicación actual con AppDomain.CurrentDomain.GetAssemblies(). Esto también funciona en .NET 5, donde es equivalente a lo siguiente:

```
AssemblyLoadContext.All.SelectMany (a => a.Assemblies)
```

Creando un Sistema de Complementos

Para demostrar completamente los conceptos que hemos cubierto en esta sección, vamos a escribir un sistema de complementos que utiliza ALCs descargables para aislar cada complemento.

Nuestro sistema de demostración inicialmente estará compuesto por tres proyectos de .NET:

Plugin.Common (library)

Define una interfaz que implementarán los complementos.

Capitalizer (library)

Un complemento que capitaliza texto.

Plugin.Host (aplicación de consola)

Localiza e invoca los complementos.

Supongamos que los proyectos residen en los siguientes directorios:

```
c:\source\PluginDemo\Plugin.Common
c:\source\PluginDemo\Capitalizer
c:\source\PluginDemo\Plugin.Host
```

Todos los proyectos harán referencia a la biblioteca Plugin.Common, y no habrá otras referencias entre los proyectos.



Si Plugin.Host hiciera referencia a Capitalizer, no estaríamos escribiendo un sistema de complementos; la idea central es que los complementos sean escritos por terceros después de que Plugin.Host y Plugin.Common hayan sido publicados.

Si estás utilizando Visual Studio, puede ser conveniente colocar los tres proyectos en una sola solución para esta demostración. Si lo haces, haz clic derecho en el proyecto Plugin.Host, elige "Build Dependencies" > "Project Dependencies" y luego selecciona el proyecto Capitalizer. Esto hará que Capitalizer se compile cuando ejecutes el proyecto Plugin.Host, sin agregar una referencia directa.

Plugin.Common

Comencemos con Plugin.Common. Nuestros complementos realizarán una tarea muy simple, que es transformar una cadena de texto. Así es como definiremos la interfaz:

```
namespace Plugin.Common
{
  public interface ITextPlugin
  {
    string TransformText (string input);
  }
}
```

Eso es todo lo que hay en Plugin.Common.

Capitalizer (complemento)

Nuestro complemento Capitalizer hará referencia a Plugin.Common y contendrá una sola clase. Por ahora, mantendremos la lógica simple para que el complemento no tenga dependencias adicionales:

```
public class CapitalizerPlugin : Plugin.Common.ITextPlugin
{
  public string TransformText (string input) => input.ToUpper();
}
```

Si compilas ambos proyectos y miras en la carpeta de salida de Capitalizer, verás las siguientes dos assemblies:

```
Capitalizer.dll      // Our plug-in assembly
Plugin.Common.dll    // Referenced assembly
```

Plugin.Host

Plugin.Host es una aplicación de consola con dos clases. La primera clase es un ALC personalizado para cargar los complementos:

```
class PluginLoadContext : AssemblyLoadContext
{
  AssemblyDependencyResolver _resolver;
```

```
  public PluginLoadContext (string pluginPath, bool collectible)
    // Give it a friendly name to help with debugging:
    : base (name: Path.GetFileName (pluginPath), collectible)
  {
    // Create a resolver to help us find dependencies.
    _resolver = new AssemblyDependencyResolver (pluginPath);
  }

  protected override Assembly Load (AssemblyName assemblyName)
  {
    // See below
    if (assemblyName.Name == typeof (ITextPlugin).Assembly.GetName().Name)
      return null;

    string target = _resolver.ResolveAssemblyToPath (assemblyName);

    if (target != null)
      return LoadFromAssemblyPath (target);

    // Could be a BCL assembly. Allow the default context to resolve.
    return null;
  }

  protected override IntPtr LoadUnmanagedDll (string unmanagedDllName)
  {
    string path = _resolver.ResolveUnmanagedDllToPath (unmanagedDllName);

    return path == null
      ? IntPtr.Zero
      : LoadUnmanagedDllFromPath (path);
  }
}
```

En el constructor, pasamos la ruta a la assembly principal del complemento, así como una bandera para indicar si deseamos que el ALC sea colectable (para que pueda ser descargado). El método Load es donde manejamos la resolución de dependencias. Todos los complementos deben hacer referencia a Plugin.Common para que puedan implementar ITextPlugin. Esto significa que en algún momento se activará el método Load para resolver Plugin.Common. Debemos tener cuidado porque es probable que la carpeta de salida del complemento contenga no solo Capitalizer.dll, sino también su propia copia de Plugin.Common.dll. Si cargáramos esta copia de Plugin.Common.dll en el PluginLoadContext, tendríamos dos copias de la assembly: una en el contexto predeterminado del host y otra en el PluginLoadContext del complemento. Las assemblies serían incompatibles y el host se quejaría de que el complemento no implementa ITextPlugin.

Para resolver esto, verificamos explícitamente esta condición:

```
if (assemblyName.Name == typeof (ITextPlugin).Assembly.GetName().Name)
  return null;
```

Devolver null permite que el ALC predeterminado del host resuelva la assembly en su lugar.



En lugar de devolver null, podríamos devolver typeof(ITextPlugin).Assembly, y también funcionaría correctamente. ¿Cómo podemos estar seguros de que ITextPlugin se resolverá en el ALC del host y no en nuestro PluginLoadContext? Recuerda que nuestra clase PluginLoadContext está definida en la assembly Plugin.Host. Por lo tanto, cualquier tipo al que hagas referencia estáticamente desde esta clase desencadenará una resolución de assembly en el ALC en el que se cargó su assembly, Plugin.Host.

Después de verificar la assembly común, usamos AssemblyDependencyResolver para localizar cualquier dependencia privada que pueda tener el complemento (por ahora, no habrá ninguna).

Observa que también anulamos el método LoadUnamangedDll. Esto asegura que si el complemento tiene alguna dependencia no administrada, esta se cargará correctamente también.

La segunda clase que debemos escribir en Plugin.Host es el programa principal en sí. Para simplificar, codifiquemos la ruta de nuestro complemento Capitalizer (en la vida real, podrías descubrir las rutas de los complementos buscando DLL en ubicaciones conocidas o leyendo desde un archivo de configuración):

```
class Program
{
  const bool UseCollectibleContexts = true;

  static void Main()
  {
    const string capitalizer = @"C:\source\PluginDemo\"
      + @"Capitalizer\bin\Debug\netcoreapp3.0\Capitalizer.dll";

    Console.WriteLine (TransformText ("big apple", capitalizer));
  }

  static string TransformText (string text, string pluginPath)
  {
    var alc = new PluginLoadContext (pluginPath, UseCollectibleContexts);
    try
    {
      Assembly assem = alc.LoadFromAssemblyPath (pluginPath);

      // Locate the type in the assembly that implements ITextPlugin:
      Type pluginType = assem.ExportedTypes.Single (t =>
                        typeof (ITextPlugin).IsAssignableFrom (t));

      // Instantiate the ITextPlugin implementation:
      var plugin = (ITextPlugin)Activator.CreateInstance (pluginType);

      // Call the TransformText method
      return plugin.TransformText (text);
    }
    finally
    {
      if (UseCollectibleContexts) alc.Unload();    // unload the ALC
    }
```

```
    }
  }
```

Veamos el método TransformText. Primero, creamos una nueva ALC para nuestro complemento y luego le pedimos que cargue la assembly principal del complemento. Luego, utilizamos Reflection para ubicar el tipo que implementa ITextPlugin (esto se explica en detalle en el Capítulo 18). A continuación, instanciamos el complemento, llamamos al método TransformText y descargamos la ALC.



Si necesitaras llamar al método TransformText repetidamente, un enfoque mejor sería almacenar en caché la ALC en lugar de descargarla después de cada llamada.

Aquí está la salida:

```
BIG APPLE
```

Agregando dependencias (Adding dependencies)

Nuestro código es completamente capaz de resolver y aislar las dependencias. Para ilustrarlo, agreguemos primero una referencia de NuGet a Humanizer.Core, versión 2.6.2. Puedes hacer esto a través de la interfaz de usuario de Visual Studio o agregando el siguiente elemento al archivo Capitalizer.csproj:

```
<ItemGroup>
  <PackageReference Include="Humanizer.Core" Version="2.6.2" />
</ItemGroup>
```

Ahora, modifiquemos CapitalizerPlugin de la siguiente manera:

```
using Humanizer;
namespace Capitalizer
{
  public class CapitalizerPlugin : Plugin.Common.ITextPlugin
  {
    public string TransformText (string input) => input.Pascalize();
  }
}
```

Si vuelves a ejecutar el programa, la salida será la siguiente:

```
BigApple
```

A continuación, crearemos otro complemento llamado Pluralizer. Crea un nuevo proyecto de biblioteca .NET y agrega una referencia de NuGet a Humanizer.Core, versión 2.7.9:

```
<ItemGroup>
  <PackageReference Include="Humanizer.Core" Version="2.7.9" />
</ItemGroup>
```

Ahora, agrega una clase llamada PluralizerPlugin. Será similar a CapitalizerPlugin, pero llamaremos al método Pluralize en su lugar:

```
using Humanizer;
namespace Pluralizer
{
  public class PluralizerPlugin : Plugin.Common.ITextPlugin

   {
     public string TransformText (string input) => input.Pluralize();
   }
 }
```

Finalmente, necesitamos agregar código al método Main de Plugin.Host para cargar y ejecutar el complemento Pluralizer:

```
static void Main()
{
  const string capitalizer = @"C:\source\PluginDemo\"
    + @"Capitalizer\bin\Debug\netcoreapp3.0\Capitalizer.dll";

  Console.WriteLine (TransformText ("big apple", capitalizer));

  const string pluralizer = @"C:\source\PluginDemo\"
    + @"Pluralizer\bin\Debug\netcoreapp3.0\Pluralizer.dll";

  Console.WriteLine (TransformText ("big apple", pluralizer));
}
```

La salida será la siguiente:

```
BigApple
big apples
```

Para ver completamente lo que está sucediendo, cambia la constante `UseCollectibleContexts` a `false` y agrega el siguiente código al método Main para enumerar las ALC y sus assemblies:

```
foreach (var context in AssemblyLoadContext.All)
{
  Console.WriteLine ($"Context: {context.GetType().Name} {context.Name}");

  foreach (var assembly in context.Assemblies)
      Console.WriteLine ($"  Assembly: {assembly.FullName}");
}
```

En la salida, podrás ver dos versiones diferentes de Humanizer, cada una cargada en su propia ALC:

```
Context: PluginLoadContext Capitalizer.dll
  Assembly: Capitalizer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=...
  Assembly: Humanizer, Version=2.6.0.0, Culture=neutral, PublicKeyToken=...
Context: PluginLoadContext Pluralizer.dll
  Assembly: Pluralizer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=...
  Assembly: Humanizer, Version=2.7.0.0, Culture=neutral, PublicKeyToken=...
Context: DefaultAssemblyLoadContext Default
  Assembly: System.Private.CoreLib, Version=4.0.0.0, Culture=neutral,...
  Assembly: Host, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
  ...
```

Incluso si ambos complementos usaran la misma versión de Humanizer, el aislamiento de las diferentes assemblies aún puede ser beneficioso porque cada una tendría sus propias variables estáticas.

18 Reflexión y Metadatos

Como vimos en el Capítulo 17, un programa en C# se compila en una assembly que incluye metadatos, código compilado y recursos. Inspeccionar los metadatos y el código compilado en tiempo de ejecución se llama reflexión.

El código compilado en una assembly contiene casi todo el contenido del código fuente original. Se pierde cierta información, como los nombres de las variables locales, los comentarios y las directivas del preprocesador. Sin embargo, la reflexión puede acceder a prácticamente todo lo demás, incluso haciendo posible escribir un descompilador.

Muchos de los servicios disponibles en .NET y expuestos a través de C# (como la vinculación dinámica, la serialización y la vinculación de datos) dependen de la presencia de metadatos. Tus propios programas también pueden aprovechar estos metadatos e incluso ampliarlos con nueva información utilizando atributos personalizados. El espacio de nombres System.Reflection alberga la API de reflexión. También es posible crear dinámicamente nuevos metadatos e instrucciones ejecutables en Intermediate Language (IL) mediante las clases del espacio de nombres System.Reflection.Emit.

Los ejemplos de este capítulo asumen que importas los espacios de nombres System y System.Reflection, así como System.Reflection.Emit.



Cuando utilizamos el término "dinámicamente" en este capítulo, nos referimos a utilizar la reflexión para realizar alguna tarea cuya seguridad de tipos se aplica solo en tiempo de ejecución. Esto es similar en principio a la vinculación dinámica mediante la palabra clave dynamic de C#, aunque el mecanismo y la funcionalidad son diferentes.

La vinculación dinámica es mucho más fácil de usar y utiliza el Dynamic Language Runtime (DLR) para la interoperabilidad de lenguajes dinámicos. La reflexión es relativamente incómoda de usar, pero es más flexible en cuanto a lo que puedes hacer con el CLR. Por ejemplo, la reflexión te permite obtener listas de tipos y miembros, instanciar un objeto cuyo nombre proviene de una cadena y construir assemblies sobre la marcha.

Reflecting and Activating Types

En esta sección, examinaremos cómo obtener un Type, inspeccionar sus metadatos y usarlo para instanciar dinámicamente un objeto.

Obtener un Type

Una instancia de System.Type representa los metadatos de un tipo. Debido a que Type se utiliza ampliamente, se encuentra en el espacio de nombres System en lugar del espacio de nombres System.Reflection.

Puedes obtener una instancia de System.Type llamando a GetType en cualquier objeto o utilizando el operador typeof en C#:

```
Type t1 = DateTime.Now.GetType();     // Type obtained at runtime
Type t2 = typeof (DateTime);          // Type obtained at compile time
```

Puedes utilizar typeof para obtener tipos de arrays y tipos genéricos de la siguiente manera:

```
Type t3 = typeof (DateTime[]);          // 1-d Array type
Type t4 = typeof (DateTime[,]);         // 2-d Array type
Type t5 = typeof (Dictionary<int,int>); // Closed generic type
Type t6 = typeof (Dictionary<,>);       // Unbound generic type
```

También puedes obtener un Type por su nombre. Si tienes una referencia a su Assembly, llama a Assembly.GetType (describiremos esto más adelante en la sección "Reflexión de Assemblies"):

```
Type t = Assembly.GetExecutingAssembly().GetType ("Demos.TestProgram");
```

Si no tienes un objeto Assembly, puedes obtener un tipo a través de su nombre cualificado por el ensamblado (el nombre completo del tipo seguido del nombre completamente o parcialmente cualificado del ensamblado). El ensamblado se carga implícitamente como si llamaras a Assembly.Load(string):

```
Type t = Type.GetType ("System.Int32, System.Private.CoreLib");
```

Una vez que tienes un objeto System.Type, puedes utilizar sus propiedades para acceder al nombre del tipo, ensamblado, tipo base, visibilidad, y más:

```
Type stringType = typeof (string);
string name     = stringType.Name;          // String
Type baseType   = stringType.BaseType;      // typeof(Object)
Assembly assem  = stringType.Assembly;      // System.Private.CoreLib
bool isPublic   = stringType.IsPublic;      // true
```

Una instancia de System.Type es una ventana a todos los metadatos del tipo y el ensamblado en el que está definido.



System.Type es abstracto, por lo que el operador typeof realmente debe darte una subclase de Type. La subclase que utiliza el CLR es interna de .NET y se llama RuntimeType.

TypeInfo

Si estás apuntando a .NET Core 1.x (o un perfil más antiguo de Windows Store), encontrarás que la mayoría de los miembros de Type están ausentes. En su lugar, estos miembros ausentes se exponen en una clase llamada TypeInfo, que se obtiene llamando a GetTypeInfo. Por lo tanto, para que nuestro ejemplo anterior funcione, harías lo siguiente:

```
Type stringType = typeof(string);
string name = stringType.Name;
Type baseType = stringType.GetTypeInfo().BaseType;
Assembly assem = stringType.GetTypeInfo().Assembly;
bool isPublic = stringType.GetTypeInfo().IsPublic;
```

TypeInfo también existe en .NET Core 2 y 3, y .NET 5+ (y .NET Framework 4.5+ y todas las versiones de .NET Standard), por lo que el código anterior funciona casi universalmente. TypeInfo también incluye propiedades y métodos adicionales para reflejar sobre miembros.

Obteniendo tipos de arrays

Como acabamos de ver, typeof y GetType funcionan con tipos de arrays. También puedes obtener un tipo de array llamando a MakeArrayType en el tipo de elemento:

```
Type simpleArrayType = typeof (int).MakeArrayType();
Console.WriteLine (simpleArrayType == typeof (int[]));     // True
```

Puedes crear arrays multidimensionales pasando un argumento entero a MakeArrayType:

```
Type cubeType = typeof (int).MakeArrayType (3);       // cube shaped
Console.WriteLine (cubeType == typeof (int[,,]));     // True
```

GetElementType hace lo contrario: recupera el tipo de elemento de un tipo de array:

```
Type e = typeof (int[]).GetElementType();     // e == typeof (int)
```

GetArrayRank devuelve el número de dimensiones de un array rectangular:

```
int rank = typeof (int[,,]).GetArrayRank();   // 3
```

Obteniendo tipos anidados

Para recuperar tipos anidados, llama a GetNestedTypes en el tipo contenedor:

```
foreach (Type t in typeof (System.Environment).GetNestedTypes())
  Console.WriteLine (t.FullName);

OUTPUT: System.Environment+SpecialFolder
```

O:

```
foreach (TypeInfo t in typeof (System.Environment).GetTypeInfo()
                                                  .DeclaredNestedTypes)
  Debug.WriteLine (t.FullName);
```

El único inconveniente con los tipos anidados es que el CLR trata a un tipo anidado como si tuviera niveles de accesibilidad "anidados" especiales:

```
Type t = typeof (System.Environment.SpecialFolder);
Console.WriteLine (t.IsPublic);                    // False
Console.WriteLine (t.IsNestedPublic);              // True
```

Nombres de tipos

Un tipo tiene propiedades de Namespace, Name y FullName. En la mayoría de los casos, FullName es una composición de las dos anteriores:

```
Type t = typeof (System.Text.StringBuilder);

Console.WriteLine (t.Namespace);      // System.Text
Console.WriteLine (t.Name);           // StringBuilder
Console.WriteLine (t.FullName);       // System.Text.StringBuilder
```

Hay dos excepciones a esta regla: los tipos anidados y los tipos genéricos cerrados.



Type también tiene una propiedad llamada AssemblyQualifiedName, que devuelve FullName seguido de una coma y luego el nombre completo de su ensamblado. Esta es la misma cadena que puedes pasar a Type.GetType, y identifica de manera única a un tipo dentro del contexto de carga predeterminado.

Nombres de tipos anidados

Con los tipos anidados, el tipo contenedor aparece solo en FullName:

```
Type t = typeof (System.Environment.SpecialFolder);

Console.WriteLine (t.Namespace);      // System
Console.WriteLine (t.Name);           // SpecialFolder
Console.WriteLine (t.FullName);       // System.Environment+SpecialFolder
```

El símbolo + diferencia el tipo contenedor de un espacio de nombres anidado.

Nombres de tipos genéricos

Los nombres de tipos genéricos tienen un sufijo de ' seguido del número de parámetros de tipo. Si el tipo genérico no está vinculado, esta regla se aplica tanto a Name como a FullName:

```
Type t = typeof (Dictionary<,>); // Unbound
Console.WriteLine (t.Name);      // Dictionary'2
Console.WriteLine (t.FullName);  // System.Collections.Generic.Dictionary'2
```

Sin embargo, si el tipo genérico está cerrado, FullName adquiere un apéndice adicional sustancial. Se enumeran los nombres completos de cada parámetro de tipo seguido de su ensamblado cualificado completo:

```
Console.WriteLine (typeof (Dictionary<int,string>).FullName);

// OUTPUT:
System.Collections.Generic.Dictionary`2[[System.Int32,

System.Private.CoreLib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=7cec85d7bea7798e],[System.String, System.Private.CoreLib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e]]
```

Esto asegura que AssemblyQualifiedName (una combinación del nombre completo del tipo y el nombre del ensamblado) contenga suficiente información para identificar completamente tanto el tipo genérico como sus parámetros de tipo.

Nombres de tipos de array y puntero

Los arrays tienen el mismo sufijo que se utiliza en una expresión typeof:

```
Console.WriteLine (typeof ( int[]  ).Name);      // Int32[]
Console.WriteLine (typeof ( int[,] ).Name);      // Int32[,]
Console.WriteLine (typeof ( int[,] ).FullName);  // System.Int32[,]
```

Los tipos de puntero son similares:

```
Console.WriteLine (typeof (byte*).Name);     // Byte*
```

Nombres de tipos de parámetros ref y out

Un Type que describe un parámetro ref o out tiene un sufijo de &:

```
public void RefMethod (ref int p)
{
  Type t = MethodInfo.GetCurrentMethod().GetParameters()[0].ParameterType;
  Console.WriteLine (t.Name);    // Int32&
}
```

Más sobre esto más adelante, en la sección "Reflexión y invocación de miembros".

Tipos base e interfaces

Type expone la propiedad BaseType:

```
Type base1 = typeof (System.String).BaseType;
Type base2 = typeof (System.IO.FileStream).BaseType;

Console.WriteLine (base1.Name);     // Object
Console.WriteLine (base2.Name);     // Stream
```

El método GetInterfaces devuelve las interfaces que implementa un tipo:

```
foreach (Type iType in typeof (Guid).GetInterfaces())
  Console.WriteLine (iType.Name);

IFormattable
IComparable
IComparable'1
IEquatable'1
```

La reflexión proporciona tres equivalentes dinámicos al operador estático is de C#:

IsInstanceOfType
Accepts a type and instance

IsAssignableFrom y (a partir de .NET 5) IsAssignableTo

Aceptan dos tipos.

Aquí tienes un ejemplo del primero:

```
object obj  = Guid.NewGuid();
Type target = typeof (IFormattable);

bool isTrue   = obj is IFormattable;             // Static C# operator
bool alsoTrue = target.IsInstanceOfType (obj);   // Dynamic equivalent
```

IsAssignableFrom es más versátil:

```
Type target = typeof (IComparable), source = typeof (string);
Console.WriteLine (target.IsAssignableFrom (source));         // True
```

El método IsSubclassOf funciona sobre el mismo principio que IsAssignableFrom, pero excluye las interfaces.

Instanciación de tipos

Hay dos formas de instanciar dinámicamente un objeto a partir de su tipo:

- Llamar al método estático Activator.CreateInstance
- Llamar a Invoke en un objeto ConstructorInfo obtenido al llamar a GetConstructor en un Type (escenarios avanzados)

Activator.CreateInstance acepta un Type y argumentos opcionales que se pasan al constructor:

```
int i = (int) Activator.CreateInstance (typeof (int));

DateTime dt = (DateTime) Activator.CreateInstance (typeof (DateTime),
                                                   2000, 1, 1);
```

CreateInstance te permite especificar muchas otras opciones, como el ensamblado desde el cual cargar el tipo y si se debe enlazar a un constructor no público. Se lanza una MissingMethodException si el tiempo de ejecución no puede encontrar un constructor adecuado.

Llamar a Invoke en un ConstructorInfo es necesario cuando los valores de los argumentos no pueden desambiguar entre constructores sobrecargados. Por ejemplo, supongamos que la clase X tiene dos constructores: uno que acepta un parámetro de tipo string y otro que acepta un parámetro de tipo StringBuilder. El objetivo es ambiguo si pasas un argumento nulo a Activator.CreateInstance. En este caso, necesitas usar un ConstructorInfo en su lugar:

```
// Fetch the constructor that accepts a single parameter of type string:
ConstructorInfo ci = typeof (X).GetConstructor (new[] { typeof (string) });

// Construct the object using that overload, passing in null:
object foo = ci.Invoke (new object[] { null });
```

O, si estás apuntando a .NET Core 1 o a un perfil anterior de Windows Store:

```
ConstructorInfo ci = typeof (X).GetTypeInfo().DeclaredConstructors
  .FirstOrDefault (c =>
     c.GetParameters().Length == 1 &&
     c.GetParameters()[0].ParameterType == typeof (string));
```

Para obtener un constructor no público, necesitas especificar BindingFlags. Consulta "Acceso a miembros no públicos" en la página 798 dentro de la sección "Reflecting and Invoking Members" en la página 789.



La instanciación dinámica agrega algunos microsegundos al tiempo que se tarda en construir el objeto. Esto es bastante en términos relativos, ya que el CLR normalmente es muy rápido en la instanciación de objetos (un simple "new" en una clase pequeña toma en la región de decenas de nanosegundos).

Para instanciar dinámicamente matrices basadas únicamente en el tipo de elemento, primero llama a MakeArrayType. También puedes instanciar tipos genéricos: lo describiremos en la siguiente sección.

Para instanciar dinámicamente un delegado, llama a Delegate.CreateDelegate. El siguiente ejemplo muestra cómo instanciar tanto un delegado de instancia como un delegado estático:

```
class Program
{
  delegate int IntFunc (int x);

  static int Square (int x) => x * x;        // Static method
  int        Cube   (int x) => x * x * x;    // Instance method

  static void Main()
  {
    Delegate staticD = Delegate.CreateDelegate
      (typeof (IntFunc), typeof (Program), "Square");

    Delegate instanceD = Delegate.CreateDelegate
      (typeof (IntFunc), new Program(), "Cube");

    Console.WriteLine (staticD.DynamicInvoke (3));      // 9
    Console.WriteLine (instanceD.DynamicInvoke (3));    // 27
  }
}
```

Puedes invocar el objeto Delegate que se devuelve llamando a DynamicInvoke, como hicimos en este ejemplo, o mediante la conversión al delegado tipado:

```
IntFunc f = (IntFunc) staticD;
Console.WriteLine (f(3));         // 9 (but much faster!)
```

Puedes pasar un MethodInfo a CreateDelegate en lugar de un nombre de método. Describiremos MethodInfo en breve, en "Reflecting and Invoking Members" en la página 789, junto con la justificación de convertir un delegado creado dinámicamente de vuelta al tipo de delegado estático.

Tipos Genéricos

Un objeto Type puede representar un tipo genérico cerrado o no ligado. Al igual que en tiempo de compilación, un tipo genérico cerrado se puede instanciar, mientras que un tipo no ligado no puede:

```
Type closed = typeof (List<int>);
List<int> list = (List<int>) Activator.CreateInstance (closed);  // OK

Type unbound   = typeof (List<>);
object anError = Activator.CreateInstance (unbound);    // Runtime error
```

El método MakeGenericType convierte un tipo no ligado en un tipo genérico cerrado. Simplemente pasa los argumentos de tipo deseados:

```
Type unbound = typeof (List<>);
Type closed = unbound.MakeGenericType (typeof (int));
```

El método GetGenericTypeDefinition hace lo contrario:

```
Type unbound2 = closed.GetGenericTypeDefinition();  // unbound == unbound2
```

La propiedad IsGenericType devuelve true si un Type es genérico, y la propiedad IsGenericTypeDefinition devuelve true si el tipo genérico es no ligado. Lo siguiente comprueba si un tipo es un tipo de valor nullable:

```
Type nullable = typeof (bool?);
Console.WriteLine (
  nullable.IsGenericType &&
  nullable.GetGenericTypeDefinition() == typeof (Nullable<>));   // True
```

El método GetGenericArguments devuelve los argumentos de tipo para los tipos genéricos cerrados:

```
Console.WriteLine (closed.GetGenericArguments()[0]);     // System.Int32
Console.WriteLine (nullable.GetGenericArguments()[0]);   // System.Boolean
```

Para los tipos genéricos no ligados, GetGenericArguments devuelve pseudotipos que representan los tipos de marcador especificados en la definición del tipo genérico:

```
Console.WriteLine (unbound.GetGenericArguments()[0]);      // T
```



En tiempo de ejecución, todos los tipos genéricos son o bien no ligados o bien cerrados. Son no ligados en el caso (relativamente inusual) de una expresión como typeof(Foo<>); de lo contrario, son cerrados. No existe tal cosa como un tipo genérico abierto en tiempo de ejecución: todos los tipos abiertos son cerrados por el compilador. El siguiente método en la clase siempre imprime False:

```
class Foo<T>
{
  public void Test()
    => Console.Write (GetType().IsGenericTypeDefinition);
}
```

Reflexionando e Invocando Miembros (Reflecting and Invoking Members)

El método GetMembers devuelve los miembros de un tipo. Considera la siguiente clase:

```
class Walnut
{
  private bool cracked;
  public void Crack() { cracked = true; }
}
```

Podemos reflejar sobre sus miembros públicos de la siguiente manera:

```
MemberInfo[] members = typeof (Walnut).GetMembers();
foreach (MemberInfo m in members)
  Console.WriteLine (m);
```

Este es el resultado:

```
Void Crack()
System.Type GetType()
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
Void .ctor()
```

Cuando se llama sin argumentos, GetMembers devuelve todos los miembros públicos de un tipo (y sus tipos base). GetMember recupera un miembro específico por nombre, aunque aún devuelve una matriz porque los miembros pueden estar sobrecargados:

```
MemberInfo[] m = typeof (Walnut).GetMember ("Crack");
Console.WriteLine (m[0]);                          // Void Crack()
```

MemberInfo también tiene una propiedad llamada MemberType de tipo MemberTypes. Esta es una enumeración de banderas con los siguientes valores:

```
All           Custom        Field         NestedType     TypeInfo
Constructor   Event         Method        Property
```

Al llamar a GetMembers, puedes pasar una instancia de MemberTypes para restringir los tipos de miembros que devuelve. Alternativamente, puedes restringir el conjunto de resultados llamando a GetMethods, GetFields, GetProperties, GetEvents, GetConstructors o GetNestedTypes. También existen versiones singulares de cada uno de estos para enfocarse en un miembro específico.



Es recomendable ser lo más específico posible al recuperar un miembro de un tipo para evitar que el código se rompa si se agregan miembros adicionales en el futuro. Si estás recuperando un método por nombre, especificar todos los tipos de parámetros asegura que tu código seguirá funcionando si el método se sobrecarga más adelante (proporcionaremos ejemplos en breve, en "Parámetros del Método" en la página 796).

Reflexionando sobre los Miembros con TypeInfo

TypeInfo expone un protocolo diferente (y algo más sencillo) para reflejar sobre los miembros. El uso de esta API es opcional (excepto en .NET Core 1 y aplicaciones antiguas de Windows Store, donde no existe un equivalente exacto al método GetMembers). En lugar de exponer métodos como GetMembers que devuelven matrices, TypeInfo expone propiedades que devuelven IEnumerable<T>, sobre las cuales generalmente se ejecutan consultas de LINQ.

La más amplia es DeclaredMembers:

```
IEnumerable<MemberInfo> members =
  typeof(Walnut).GetTypeInfo().DeclaredMembers;
```

A diferencia de GetMembers(), el resultado excluye los miembros heredados:

```
Void Crack()
Void .ctor()
Boolean cracked
```

También existen propiedades para devolver tipos específicos de miembros (DeclaredProperties, DeclaredMethods, DeclaredEvents, etc.) y métodos para devolver un miembro específico por nombre (por ejemplo, GetDeclaredMethod). Este último no se puede utilizar en métodos sobrecargados (porque no hay forma de especificar los tipos de parámetros). En su lugar, se puede ejecutar una consulta de LINQ sobre DeclaredMethods:

```
MethodInfo method = typeof (int).GetTypeInfo().DeclaredMethods
  .FirstOrDefault (m => m.Name == "ToString" &&
                        m.GetParameters().Length == 0);
```

Un objeto MemberInfo tiene una propiedad Name y dos propiedades de tipo Type:

DeclaringType

Devuelve el Type que define el miembro.

ReflectedType

Devuelve el Type sobre el cual se llamó a GetMembers.

Estas dos propiedades difieren cuando se llaman en un miembro que está definido en un tipo base: DeclaringType devuelve el tipo base, mientras que ReflectedType devuelve el subtipo. El siguiente ejemplo muestra esto:

```
// MethodInfo is a subclass of MemberInfo; see Figure 18-1.

MethodInfo test = typeof (Program).GetMethod ("ToString");
MethodInfo obj  = typeof (object) .GetMethod ("ToString");

Console.WriteLine (test.DeclaringType);      // System.Object
Console.WriteLine (obj.DeclaringType);       // System.Object

Console.WriteLine (test.ReflectedType);      // Program
Console.WriteLine (obj.ReflectedType);       // System.Object

Console.WriteLine (test == obj);             // False
```

Debido a que tienen diferentes ReflectedTypes, los objetos "test" y "obj" no son iguales. Sin embargo, esta diferencia es puramente una creación de la API de reflexión; nuestro tipo Program no tiene un método ToString distinto en el sistema de tipos subyacente. Podemos verificar que los dos objetos MethodInfo se refieren al mismo método de dos formas:

```
Console.WriteLine (test.MethodHandle == obj.MethodHandle);    // True

Console.WriteLine (test.MetadataToken == obj.MetadataToken    // True
                   && test.Module == obj.Module);
```

Un MethodHandle es único para cada método (genuinamente distinto) dentro de un proceso; un MetadataToken es único en todos los tipos y miembros dentro de un módulo de ensamblado.

MemberInfo también define métodos para obtener atributos personalizados (ver "Obtención de atributos en tiempo de ejecución" en la página 807).



Puedes obtener el MethodBase del método que se está ejecutando actualmente llamando a MethodBase.GetCurrentMethod.

Tipos de miembros

MemberInfo en sí mismo tiene pocos miembros porque es una clase base abstracta para los tipos mostrados en la Figura 18-1.

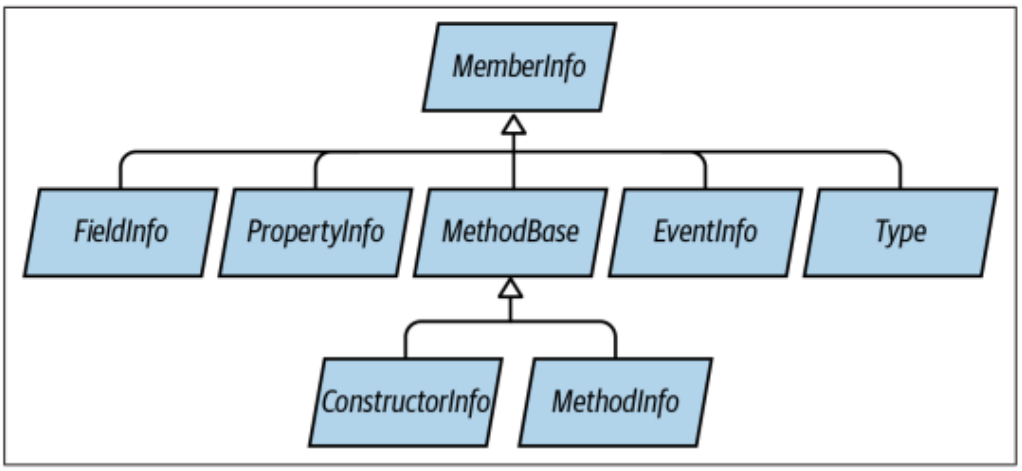


Figure 18-1. Member types

Puedes hacer un casting de un MemberInfo a su subtipo correspondiente, basándote en la propiedad MemberType. Si obtuviste un miembro a través de GetMethod, GetField, GetProperty, GetEvent, GetConstructor o GetNestedType (o sus versiones en plural), no es necesario hacer un casting. La Tabla 18-1 resume qué métodos utilizar para cada tipo de construcción en C#.

Table 18-1. Retrieving member metadata

C# construct	Method to use	Name to use	Result
Method	`GetMethod`	(method name)	`MethodInfo`
Property	`GetProperty`	(property name)	`PropertyInfo`
Indexer	`GetDefaultMembers`		`MemberInfo[]` (containing `PropertyInfo` objects if compiled in C#)
Field	`GetField`	(field name)	`FieldInfo`
Enum member	`GetField`	(member name)	`FieldInfo`
Event	`GetEvent`	(event name)	`EventInfo`
Constructor	`GetConstructor`		`ConstructorInfo`
Finalizer	`GetMethod`	`"Finalize"`	`MethodInfo`
Operator	`GetMethod`	`"op_"` + operator name	`MethodInfo`
Nested type	`GetNestedType`	(type name)	`Type`

Cada subclase de MemberInfo tiene una amplia variedad de propiedades y métodos que exponen todos los aspectos de los metadatos del miembro. Esto incluye elementos como visibilidad, modificadores, argumentos de tipo genérico, parámetros, tipo de retorno y atributos personalizados.

Aquí tienes un ejemplo de uso de GetMethod:

```
MethodInfo m = typeof (Walnut).GetMethod ("Crack");
Console.WriteLine (m);                              // Void Crack()
Console.WriteLine (m.ReturnType);                   // System.Void
```

Todas las instancias de *Info se almacenan en caché por la API de reflexión en su primer uso:

```
MethodInfo method = typeof (Walnut).GetMethod ("Crack");
MemberInfo member = typeof (Walnut).GetMember ("Crack") [0];

Console.Write (method == member);       // True
```

Además de preservar la identidad del objeto, la caché mejora el rendimiento de lo que, de lo contrario, sería una API bastante lenta.

Miembros de C# versus Miembros de CLR

La tabla anterior ilustra que algunos de los constructos funcionales de C# no tienen una correspondencia 1:1 con los constructos de CLR. Esto tiene sentido porque el CLR y la API de reflexión fueron diseñados teniendo en cuenta todos los lenguajes de .NET, por lo que incluso se puede utilizar la reflexión desde Visual Basic y otros lenguajes.

Algunos constructos de C#, como los indexadores, los enumeradores, los operadores y los finalizadores, son trivialidades en lo que respecta al CLR (Common Language Runtime). Específicamente:

- Un indexador de C# se traduce en una propiedad que acepta uno o más argumentos, marcada como el [DefaultMember] del tipo.
- Un enumerador de C# se traduce en un subtipo de System.Enum con un campo estático para cada miembro.
- Un operador de C# se traduce en un método estático con un nombre especial que comienza con "op_", por ejemplo, "op_Addition".
- Un finalizador de C# se traduce en un método que anula el método Finalize.

Otra complicación es que las propiedades y los eventos en realidad comprenden dos elementos:

- Metadatos que describen la propiedad o el evento (encapsulados por PropertyInfo o EventInfo).
- Uno o dos métodos de respaldo.

En un programa de C#, los métodos de respaldo están encapsulados dentro de la definición de la propiedad o el evento. Pero cuando se compila a IL (Intermediate Language), los métodos de respaldo aparecen como métodos ordinarios que se pueden llamar como cualquier otro. Esto significa que GetMethods devuelve los métodos de respaldo de propiedades y eventos junto con los métodos ordinarios:

```
class Test { public int X { get { return 0; } set {} } }

void Demo()
{
  foreach (MethodInfo mi in typeof (Test).GetMethods())
    Console.Write (mi.Name + "  ");
}

// OUTPUT:
get_X  set_X  GetType  ToString  Equals  GetHashCode
```

Aquí tienes un ejemplo de cómo identificar estos métodos a través de la propiedad IsSpecialName en MethodInfo. IsSpecialName devuelve true para los accesores de propiedades, indexadores y eventos, así como para los operadores. Devuelve false solo para los métodos convencionales de C# y el método Finalize si se define un finalizador.

Aquí están los métodos de respaldo que genera C#:

C# construct	Member type	Methods in IL
Property	`Property`	`get_`*XXX* and `set_`*XXX*
Indexer	`Property`	`get_Item` and `set_Item`
Event	`Event`	`add_`*XXX* and `remove_`*XXX*

Cada método de respaldo tiene su propio objeto MethodInfo asociado. Puedes acceder a ellos de la siguiente manera:

```
PropertyInfo pi = typeof (Console).GetProperty ("Title");
MethodInfo getter = pi.GetGetMethod();                    // get_Title
MethodInfo setter = pi.GetSetMethod();                    // set_Title
MethodInfo[] both = pi.GetAccessors();                    // Length==2
```

Los métodos GetAddMethod y GetRemoveMethod realizan un trabajo similar para EventInfo.

Para ir en la dirección opuesta, es decir, desde un objeto MethodInfo hasta su PropertyInfo o EventInfo asociado, necesitas realizar una consulta. LINQ es ideal para este trabajo. Por ejemplo:

```
PropertyInfo p = mi.DeclaringType.GetProperties()
                   .First (x => x.GetAccessors (true).Contains (mi));
```

Propiedades de solo inicialización (Init-only properties):

Las propiedades de solo inicialización, introducidas en C# 9, pueden ser establecidas mediante un inicializador de objetos, pero posteriormente son tratadas como de solo lectura por el compilador. Desde la perspectiva del CLR, un accesor de inicialización (init) es similar a un accesor de escritura (set) ordinario, pero con un indicador especial aplicado al tipo de retorno del método set (lo cual tiene un significado para el compilador).

Curiosamente, este indicador no se codifica como un atributo de convención. En cambio, utiliza un mecanismo relativamente oscuro llamado modreq, que garantiza que las versiones anteriores del compilador de C# (que no reconocen el nuevo modreq) ignoren el accesor en lugar de tratar la propiedad como escribible.

El modreq para los accesores de solo inicialización se llama IsExternalInit, y puedes consultarlo de la siguiente manera:

```
bool IsInitOnly (PropertyInfo pi) => pi
  .GetSetMethod().ReturnParameter.GetRequiredCustomModifiers()
  .Any (t => t.Name == "IsExternalInit");
```

NullabilityContextInfo

A partir de .NET 6, puedes utilizar la clase NullabilityInfoContext para obtener información sobre las anotaciones de nulabilidad para un campo, propiedad, evento o parámetro:

```
void PrintPropertyNullability (PropertyInfo pi)
{
  var info = new NullabilityInfoContext().Create (pi);
  Console.WriteLine (pi.Name + " read " + info.ReadState);
  Console.WriteLine (pi.Name + " write " + info.WriteState);
  // Use info.Element to get nullability info for array elements
}
```

Miembros de tipos genéricos

Puedes obtener metadatos de miembros tanto para tipos genéricos sin ligar como para tipos genéricos cerrados:

```
PropertyInfo unbound = typeof (IEnumerator<>)  .GetProperty ("Current");
PropertyInfo closed = typeof (IEnumerator<int>).GetProperty ("Current");


Console.WriteLine (unbound);   // T Current
Console.WriteLine (closed);    // Int32 Current

Console.WriteLine (unbound.PropertyType.IsGenericParameter);  // True
Console.WriteLine (closed.PropertyType.IsGenericParameter);   // False
```

Los objetos MemberInfo devueltos de tipos genéricos sin ligar y tipos genéricos cerrados son siempre distintos, incluso para miembros cuyas firmas no incluyen parámetros de tipo genérico:

```
PropertyInfo unbound = typeof (List<>)  .GetProperty ("Count");
PropertyInfo closed = typeof (List<int>).GetProperty ("Count");

Console.WriteLine (unbound);   // Int32 Count
Console.WriteLine (closed);    // Int32 Count

Console.WriteLine (unbound == closed);   // False

Console.WriteLine (unbound.DeclaringType.IsGenericTypeDefinition); // True
Console.WriteLine (closed.DeclaringType.IsGenericTypeDefinition); // False
```

Los miembros de tipos genéricos sin ligar no pueden ser invocados dinámicamente.

Invocando Dinámicamente un Miembro (Dynamically Invoking a Member)

Para invocar dinámicamente un miembro, puedes hacerlo más fácilmente utilizando la biblioteca de código abierto Uncapsulator (disponible en NuGet y GitHub). Uncapsulator fue escrita por el autor y proporciona una API fluida para invocar miembros públicos y no públicos mediante reflexión, utilizando un enlace dinámico personalizado.

Una vez que tienes un objeto MethodInfo, PropertyInfo o FieldInfo, puedes llamarlo dinámicamente o obtener/establecer su valor. Esto se conoce como enlace tardío (late binding)

porque eliges qué miembro invocar en tiempo de ejecución en lugar de en tiempo de compilación.

Para ilustrar, lo siguiente utiliza enlace estático ordinario:

```
string s = "Hello";
int length = s.Length;
```

Aquí está lo mismo realizado dinámicamente con enlace tardío:

```
object s = "Hello";
PropertyInfo prop = s.GetType().GetProperty ("Length");
int length = (int) prop.GetValue (s, null);              // 5
```

GetValue y SetValue obtienen y establecen el valor de un PropertyInfo o FieldInfo. El primer argumento es la instancia, que puede ser nulo para un miembro estático. Acceder a un indexador es similar a acceder a una propiedad llamada "Item", excepto que debes proporcionar los valores del indexador como segundo argumento al llamar a GetValue o SetValue.

Para llamar dinámicamente a un método, utiliza el método Invoke en un objeto MethodInfo, proporcionando un arreglo de argumentos para pasar a ese método. Si alguno de los tipos de argumentos es incorrecto, se lanzará una excepción en tiempo de ejecución. Con la invocación dinámica, se pierde la seguridad de tipos en tiempo de compilación, pero aún se mantiene la seguridad de tipos en tiempo de ejecución (al igual que con la palabra clave dynamic).

Parámetros del Método

Supongamos que queremos llamar dinámicamente al método Substring de la clase string. Estáticamente, lo haríamos de la siguiente manera:

```
Console.WriteLine ("stamp".Substring(2));                // "amp"
```

Aquí está el equivalente dinámico con reflexión y enlace tardío:

```
Type type = typeof (string);
Type[] parameterTypes = { typeof (int) };
MethodInfo method = type.GetMethod ("Substring", parameterTypes);

object[] arguments = { 2 };
object returnValue = method.Invoke ("stamp", arguments);
Console.WriteLine (returnValue);                         // "amp"
```

Debido a que el método Substring está sobrecargado, tuvimos que pasar un arreglo de tipos de parámetros a GetMethod para indicar qué versión queríamos. Sin los tipos de parámetros, GetMethod lanzaría una AmbiguousMatchException.

El método GetParameters, definido en MethodBase (la clase base para MethodInfo y ConstructorInfo), devuelve metadatos de parámetros. Podemos continuar con nuestro ejemplo anterior de la siguiente manera:

```
ParameterInfo[] paramList = method.GetParameters();
foreach (ParameterInfo x in paramList)
{
  Console.WriteLine (x.Name);                // startIndex
  Console.WriteLine (x.ParameterType);       // System.Int32
}
```

Tratando con parámetros ref y out

Para pasar parámetros ref o out, llama a MakeByRefType en el tipo antes de obtener el método. Por ejemplo, puedes ejecutar dinámicamente el siguiente código:

```
int x;
bool successfulParse = int.TryParse ("23", out x);
```

de la siguiente manera:

```
object[] args = { "23", 0 };
Type[] argTypes = { typeof (string), typeof (int).MakeByRefType() };
MethodInfo tryParse = typeof (int).GetMethod ("TryParse", argTypes);
bool successfulParse = (bool) tryParse.Invoke (null, args);

Console.WriteLine (successfulParse + " " + args[1]);       // True 23
```

Este enfoque funciona tanto para parámetros ref como para parámetros out.

Recuperar e invocar métodos genéricos

Especificar explícitamente los tipos de parámetros al llamar al método GetMethod puede ser esencial para desambiguar métodos sobrecargados. Sin embargo, es imposible especificar tipos de parámetros genéricos. Por ejemplo, considera la clase System.Linq.Enumerable, la cual sobrecarga el método Where de la siguiente manera:

```
public static IEnumerable<TSource> Where<TSource>
 (this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static IEnumerable<TSource> Where<TSource>
 (this IEnumerable<TSource> source, Func<TSource, int, bool> predicate);
```

Para recuperar una sobrecarga específica, debemos obtener todos los métodos y luego encontrar manualmente la sobrecarga deseada. La siguiente consulta recupera la primera sobrecarga de Where:

```
from m in typeof (Enumerable).GetMethods()
where m.Name == "Where" && m.IsGenericMethod
let parameters = m.GetParameters()
where parameters.Length == 2
let genArg = m.GetGenericArguments().First()
let enumerableOfT = typeof (IEnumerable<>).MakeGenericType (genArg)
let funcOfTBool = typeof (Func<,>).MakeGenericType (genArg, typeof (bool))
where parameters[0].ParameterType == enumerableOfT
   && parameters[1].ParameterType == funcOfTBool
select m
```

Llamar a .Single() en esta consulta devuelve el objeto MethodInfo correcto con parámetros de tipo no ligados. El siguiente paso es cerrar los parámetros de tipo mediante el método MakeGenericMethod:

```
var closedMethod = unboundMethod.MakeGenericMethod (typeof (int));
```

En este caso, hemos cerrado TSource con int, lo que nos permite llamar a Enumerable.Where con una fuente de tipo IEnumerable<int> y un predicado de tipo Func<int,bool>:

```
int[] source = { 3, 4, 5, 6, 7, 8 };
Func<int, bool> predicate = n => n % 2 == 1;   // Odd numbers only
```

Ahora podemos invocar el método genérico cerrado:

```
var query = (IEnumerable<int>) closedMethod.Invoke
  (null, new object[] { source, predicate });

foreach (int element in query) Console.Write (element + "|");  // 3|5|7|
```



Si estás utilizando la API System.Linq.Expressions para construir expresiones dinámicamente (Capítulo 8), no necesitas complicarte tanto para especificar un método genérico. El método Expression.Call tiene sobrecargas que te permiten especificar los argumentos de tipo cerrado del método que deseas llamar:

```
int[] source = { 3, 4, 5, 6, 7, 8 };
Func<int, bool> predicate = n => n % 2 == 1;

var sourceExpr = Expression.Constant (source);
var predicateExpr = Expression.Constant (predicate);

var callExpression = Expression.Call (
  typeof (Enumerable), "Where",
  new[] { typeof (int) },  // Closed generic arg type.
  sourceExpr, predicateExpr);
```

Uso de delegados para mejorar el rendimiento

Las invocaciones dinámicas son relativamente ineficientes, con un sobrecosto típicamente en el rango de unos pocos microsegundos. Si estás llamando a un método repetidamente en un bucle, puedes reducir el sobrecosto por llamada al rango de nanosegundos llamando en su lugar a un delegado instanciado dinámicamente que apunte a tu método dinámico. En el siguiente ejemplo, llamamos dinámicamente al método Trim de la cadena un millón de veces sin un sobrecosto significativo:

```
MethodInfo trimMethod = typeof (string).GetMethod ("Trim", new Type[0]);
var trim = (StringToString) Delegate.CreateDelegate
                                    (typeof (StringToString), trimMethod);
for (int i = 0; i < 1000000; i++)
  trim ("test");

delegate string StringToString (string s);
```

Esto es más rápido porque el enlace tardío costoso (mostrado en negrita) ocurre solo una vez.

Acceder a miembros no públicos

Todos los métodos en los tipos utilizados para sondear metadatos (por ejemplo, GetProperty, GetField, etc.) tienen sobrecargas que toman un enum BindingFlags. Este enum sirve como un filtro de metadatos y te permite cambiar los criterios de selección predeterminados. El uso más común para esto es recuperar miembros no públicos (esto solo funciona en aplicaciones de escritorio).

Por ejemplo, considera la siguiente clase:

```
class Walnut
{
  private bool cracked;
  public void Crack() { cracked = true; }

  public override string ToString() { return cracked.ToString(); }
}
```

Podemos resolver el problema de la siguiente manera:

```
Type t = typeof (Walnut);
Walnut w = new Walnut();

w.Crack();
FieldInfo f = t.GetField ("cracked", BindingFlags.NonPublic |
                                     BindingFlags.Instance);
f.SetValue (w, false);
Console.WriteLine (w);         // False
```

El uso de reflexión para acceder a miembros no públicos es poderoso, pero también es peligroso porque puedes evadir la encapsulación, creando una dependencia inmanejable en la implementación interna de un tipo.

El enum BindingFlags

BindingFlags está destinado a combinarse de forma bit a bit. Para obtener cualquier coincidencia, debes comenzar con una de las siguientes cuatro combinaciones:

```
BindingFlags.Public    | BindingFlags.Instance
BindingFlags.Public    | BindingFlags.Static
BindingFlags.NonPublic | BindingFlags.Instance
BindingFlags.NonPublic | BindingFlags.Static
```

NonPublic incluye internal, protected, protected internal y private.

El siguiente ejemplo recupera todos los miembros estáticos públicos del tipo object:

```
BindingFlags publicStatic = BindingFlags.Public | BindingFlags.Static;
MemberInfo[] members = typeof (object).GetMembers (publicStatic);
```

El siguiente ejemplo recupera todos los miembros no públicos del tipo object, tanto estáticos como de instancia:

```
BindingFlags nonPublicBinding =
  BindingFlags.NonPublic | BindingFlags.Static | BindingFlags.Instance;

MemberInfo[] members = typeof (object).GetMembers (nonPublicBinding);
```

La bandera DeclaredOnly excluye las funciones heredadas de tipos base, a menos que se anulen.



La bandera DeclaredOnly es un poco confusa porque restringe el conjunto de resultados (mientras que todas las demás banderas de enlace amplían el conjunto de resultados).

Métodos genéricos

No puedes invocar directamente métodos genéricos; lo siguiente arroja una excepción:

```
class Program
{
  public static T Echo<T> (T x) { return x; }

  static void Main()
  {
    MethodInfo echo = typeof (Program).GetMethod ("Echo");
    Console.WriteLine (echo.IsGenericMethodDefinition);    // True
    echo.Invoke (null, new object[] { 123 } );             // Exception


   }
 }
```

Se requiere un paso adicional, que es llamar a MakeGenericMethod en el MethodInfo, especificando los argumentos de tipo genérico concretos. Esto devuelve otro MethodInfo, que luego puedes invocar de la siguiente manera:

```
MethodInfo echo = typeof (Program).GetMethod ("Echo");
MethodInfo intEcho = echo.MakeGenericMethod (typeof (int));
Console.WriteLine (intEcho.IsGenericMethodDefinition);           // False
Console.WriteLine (intEcho.Invoke (null, new object[] { 3 } ));  // 3
```

Llamar de forma anónima a miembros de una interfaz genérica

La reflexión es útil cuando necesitas invocar un miembro de una interfaz genérica y no conoces los parámetros de tipo hasta tiempo de ejecución. En teoría, la necesidad de esto surge raramente si los tipos están perfectamente diseñados; por supuesto, los tipos no siempre están perfectamente diseñados.

Por ejemplo, supongamos que queremos escribir una versión más potente de ToString que pueda expandir el resultado de consultas LINQ. Podríamos comenzar de la siguiente manera:

```
public static string ToStringEx <T> (IEnumerable<T> sequence)
{
  ...
}
```

Esto ya es bastante limitante. ¿Qué sucede si sequence contiene colecciones anidadas que también queremos enumerar? Necesitaríamos sobrecargar el método para manejarlo:

```
public static string ToStringEx <T> (IEnumerable<IEnumerable<T>> sequence)
```

Y luego, ¿qué sucede si sequence contiene agrupamientos o proyecciones de secuencias anidadas? La solución estática de la sobrecarga de métodos se vuelve impracticable; necesitamos un enfoque que pueda escalar para manejar un gráfico de objetos arbitrario, como el siguiente:

```
public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  StringBuilder sb = new StringBuilder();

  if (value is List<>)                                          // Error
    sb.Append ("List of " + ((List<>) value).Count + " items");  // Error

  if (value is IGrouping<,>)                                    // Error
    sb.Append ("Group with key=" + ((IGrouping<,>) value).Key);  // Error

  // Enumerate collection elements if this is a collection,
  // recursively calling ToStringEx()
  // ...

  return sb.ToString();
}
```

Desafortunadamente, esto no se compilará: no puedes invocar miembros de un tipo genérico sin enlazar como List<> o IGrouping<>. En el caso de List<>, podemos resolver el problema utilizando la interfaz no genérica IList en su lugar:

```
if (value is IList)
  sb.AppendLine ("A list with " + ((IList) value).Count + " items");
```



Podemos hacer esto porque los diseñadores de List<> tuvieron la previsión de implementar IList clásico (así como IList genérico).

El mismo principio es digno de consideración al escribir tus propios tipos genéricos: tener una interfaz no genérica o una clase base a la cual los consumidores puedan recurrir puede ser extremadamente valioso.

La solución no es tan simple para IGrouping<,>. Así es cómo se define la interfaz:

```
public interface IGrouping <TKey,TElement> : IEnumerable <TElement>,
                                             IEnumerable
{
  TKey Key { get; }
}
```

No hay un tipo no genérico que podamos utilizar para acceder a la propiedad Key, por lo que aquí debemos utilizar la reflexión. La solución no es invocar miembros de un tipo genérico sin enlazar (lo cual es imposible), sino invocar miembros de un tipo genérico cerrado, cuyos argumentos de tipo establecemos en tiempo de ejecución.



En el siguiente capítulo, resolvemos esto de manera más simple con la palabra clave dynamic de C#. Una buena indicación para el enlace dinámico es cuando, de lo contrario, tendrías que realizar malabarismos con los tipos, como estamos haciendo ahora mismo.

El primer paso es determinar si value implementa IGrouping<,> y, de ser así, obtener su interfaz genérica cerrada. Podemos hacer esto de manera más fácil ejecutando una consulta LINQ. Luego, recuperamos e invocamos la propiedad Key:

```
public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  if (value.GetType().IsPrimitive) return value.ToString();

  StringBuilder sb = new StringBuilder();

  if (value is IList)
    sb.Append ("List of " + ((IList)value).Count + " items: ");

  Type closedIGrouping = value.GetType().GetInterfaces()
    .Where (t => t.IsGenericType &&
                 t.GetGenericTypeDefinition() == typeof (IGrouping<,>))
    .FirstOrDefault();

  if (closedIGrouping != null)   // Call the Key property on IGrouping<,>
  {
    PropertyInfo pi = closedIGrouping.GetProperty ("Key");

    object key = pi.GetValue (value, null);
    sb.Append ("Group with key=" + key + ": ");
  }

  if (value is IEnumerable)
    foreach (object element in ((IEnumerable)value))
      sb.Append (ToStringEx (element) + " ");

  if (sb.Length == 0) sb.Append (value.ToString());

  return "\r\n" + sb.ToString();
}
```

Este enfoque es robusto: funciona tanto si IGrouping<,> se implementa de forma implícita como explícita. Lo siguiente demuestra este método:

```
Console.WriteLine (ToStringEx (new List<int> { 5, 6, 7 } ));
Console.WriteLine (ToStringEx ("xyyzzz".GroupBy (c => c) ));

List of 3 items: 5 6 7

Group with key=x: x
Group with key=y: y y
Group with key=z: z z z
```

Reflejando ensamblados

Puedes reflejar dinámicamente un ensamblado llamando a GetType o GetTypes en un objeto Assembly. Lo siguiente recupera del ensamblado actual el tipo llamado "TestProgram" en el espacio de nombres Demos:

```
Type t = Assembly.GetExecutingAssembly().GetType ("Demos.TestProgram");
```

También puedes obtener un ensamblado a partir de un tipo existente:

```
typeof (Foo).Assembly.GetType ("Demos.TestProgram");
```

El siguiente ejemplo enumera todos los tipos en el ensamblado mylib.dll en e:\demo:

```
Assembly a = Assembly.LoadFile (@"e:\demo\mylib.dll");

foreach (Type t in a.GetTypes())
  Console.WriteLine (t);
```

O también:

```
Assembly a = typeof (Foo).GetTypeInfo().Assembly;

foreach (Type t in a.ExportedTypes)
  Console.WriteLine (t);
```

GetTypes y ExportedTypes solo devuelven tipos de nivel superior y no tipos anidados.

Módulos

Llamar a GetTypes en un ensamblado multimódulo devuelve todos los tipos en todos los módulos. Como resultado, puedes ignorar la existencia de los módulos y tratar un ensamblado como un contenedor de tipos. Sin embargo, hay un caso en el que los módulos son relevantes, y eso es cuando se trabaja con tokens de metadatos.

Un token de metadatos es un entero que se refiere de manera única a un tipo, miembro, cadena o recurso dentro del ámbito de un módulo. El lenguaje intermedio (IL) utiliza tokens de metadatos, por lo que si estás analizando IL, necesitarás poder resolverlos. Los métodos para hacer esto están definidos en el tipo Module y se llaman ResolveType, ResolveMember,

ResolveString y ResolveSignature. Volvemos a esto en la sección final de este capítulo, sobre cómo escribir un desensamblador.

Puedes obtener una lista de todos los módulos en un ensamblado llamando a GetModules. También puedes acceder al módulo principal de un ensamblado directamente a través de su propiedad ManifestModule.

Trabajando con atributos

El CLR permite adjuntar metadatos adicionales a tipos, miembros y ensamblados a través de atributos. Este es el mecanismo mediante el cual muchas funciones del CLR, como la serialización y la seguridad, se dirigen, lo que hace que los atributos sean una parte indivisible de una aplicación.

Una característica clave de los atributos es que puedes escribir los tuyos propios y luego usarlos de la misma manera que cualquier otro atributo para "decorar" un elemento de código con información adicional. Esta información adicional se compila en el ensamblado subyacente y se puede recuperar en tiempo de ejecución utilizando reflexión para construir servicios que funcionen de manera declarativa, como pruebas unitarias automatizadas.

Conceptos básicos de los atributos

Existen tres tipos de atributos:

- Atributos bit-mapeados
- Atributos personalizados
- Atributos seudopersonalizados

De estos, solo los atributos personalizados son extensibles.



El término "atributo" por sí mismo puede referirse a cualquiera de los tres, aunque en el mundo de C#, generalmente se refiere a atributos personalizados o atributos seudopersonalizados.

Los atributos bit-mapeados (nuestra terminología) se asignan a bits dedicados en los metadatos de un tipo. La mayoría de las palabras clave de modificación de C#, como public, abstract y sealed, se compilan como atributos bit-mapeados. Estos atributos son muy eficientes porque consumen un espacio mínimo en los metadatos (generalmente solo un bit) y el CLR puede localizarlos y trabajar con ellos con poco o ningún nivel de indirección. La API de reflexión los expone a través de propiedades dedicadas en Type (y otras subclases de MemberInfo), como IsPublic, IsAbstract e IsSealed. La propiedad Attributes devuelve un enum de banderas que describe la mayoría de ellos de una vez:

```
static void Main()
{
  TypeAttributes ta = typeof (Console).Attributes;
  MethodAttributes ma = MethodInfo.GetCurrentMethod().Attributes;
  Console.WriteLine (ta + "\r\n" + ma);
}
```

Aquí está el resultado:

```
AutoLayout, AnsiClass, Class, Public, Abstract, Sealed, BeforeFieldInit
PrivateScope, Private, Static, HideBySig
```

En contraste, los atributos personalizados se compilan en un blob que se vincula a la tabla de metadatos principal del tipo. Todos los atributos personalizados están representados por una subclase de System.Attribute y, a diferencia de los atributos bit-mapeados, son extensibles. El blob en los metadatos identifica la clase de atributo y también almacena los valores de cualquier argumento posicional o con nombre que se haya especificado al aplicar el atributo. Los atributos personalizados que defines tú mismo son arquitectónicamente idénticos a los definidos en las bibliotecas .NET.

El Capítulo 4 describió cómo adjuntar atributos personalizados a un tipo o miembro en C#. Aquí adjuntamos el atributo predefinido Obsolete a la clase Foo:

```
[Obsolete] public class Foo {...}
```

Esto le indica al compilador que incorpore una instancia de ObsoleteAttribute en los metadatos de Foo, que luego se puede reflejar en tiempo de ejecución llamando a GetCustomAttributes en un objeto Type o MemberInfo.

Los atributos pseudopersonalizados se ven y se sienten como atributos personalizados estándar. Están representados por una subclase de System.Attribute y se adjuntan de la manera estándar:

```
[Serializable] public class Foo {...}
```

La diferencia es que el compilador o CLR optimiza internamente los atributos pseudopersonalizados convirtiéndolos en atributos bit-mapeados. Ejemplos de ello son [Serializable], StructLayout, In y Out (Capítulo 24). La reflexión expone los atributos pseudopersonalizados a través de propiedades dedicadas como IsSerializable, y en muchos casos también se devuelven como objetos System.Attribute cuando se llama a GetCustomAttributes (incluido SerializableAttribute). Esto significa que puedes (casi) ignorar la diferencia entre atributos pseudopersonalizados y no pseudopersonalizados (una excepción notable es cuando se utiliza Reflection.Emit para generar tipos dinámicamente en tiempo de ejecución; consulta "Generando ensamblados y tipos" en la página 815).

El atributo AttributeUsage

AttributeUsage es un atributo aplicado a clases de atributos. Instruye al compilador sobre cómo se debe usar el atributo de destino:

```
public sealed class AttributeUsageAttribute : Attribute
{
  public AttributeUsageAttribute (AttributeTargets validOn);

  public bool AllowMultiple        { get; set; }
  public bool Inherited            { get; set; }
  public AttributeTargets ValidOn  { get; }
}
```

AllowMultiple controla si el atributo definido puede aplicarse más de una vez al mismo destino; Inherited controla si un atributo aplicado a una clase base también se aplica a las clases derivadas (o, en el caso de métodos, si un atributo aplicado a un método virtual también se aplica a los métodos de anulación). ValidOn determina el conjunto de destinos (clases, interfaces, propiedades, métodos, parámetros, etc.) a los que se puede adjuntar el atributo. Acepta cualquier combinación de valores de la enumeración AttributeTargets, que tiene los siguientes miembros:

```
All          Delegate  GenericParameter  Parameter
Assembly     Enum      Interface         Property
Class        Event     Method            ReturnValue
Constructor  Field     Module            Struct
```

Para ilustrar, así es como los autores de .NET han aplicado AttributeUsage al atributo Serializable:

```
[AttributeUsage (AttributeTargets.Delegate |
                 AttributeTargets.Enum     |
                 AttributeTargets.Struct   |
                 AttributeTargets.Class,     Inherited = false)
]
public sealed class SerializableAttribute : Attribute { }
```

De hecho, esta es casi la definición completa del atributo Serializable. Escribir una clase de atributo que no tenga propiedades o constructores especiales es tan simple como esto.

Definir tu propio atributo

Aquí tienes cómo escribir tu propio atributo:

1. Deriva una clase de System.Attribute o un descendiente de System.Attribute. Por convención, el nombre de la clase debe terminar con la palabra "Attribute", aunque esto no es obligatorio.

2. Aplica el atributo AttributeUsage, descrito en la sección anterior. Si el atributo no requiere propiedades ni argumentos en su constructor, entonces el trabajo está terminado.

3. Escribe uno o más constructores públicos. Los parámetros del constructor definen los parámetros posicionales del atributo y se volverán obligatorios al usar el atributo.

4. Declara un campo o propiedad pública para cada parámetro con nombre que desees admitir. Los parámetros con nombre son opcionales al usar el atributo.



Las propiedades de atributo y los parámetros del constructor deben ser de los siguientes tipos:

- Un tipo primitivo sellado: es decir, bool, byte, char, double, float, int, long, short o string.
- El tipo Type.
- Un tipo enum.
- Un array unidimensional de cualquiera de estos tipos.

Cuando se aplica un atributo, también debe ser posible que el compilador evalúe estáticamente cada una de las propiedades o argumentos del constructor.

La siguiente clase define un atributo para ayudar a un sistema de pruebas unitarias automatizado. Indica que un método debe ser probado, el número de repeticiones de la prueba y un mensaje en caso de fallo:

```
[AttributeUsage (AttributeTargets.Method)]
public sealed class TestAttribute : Attribute
{
  public int     Repetitions;
  public string  FailureMessage;

  public TestAttribute () : this (1)     { }
  public TestAttribute (int repetitions) { Repetitions = repetitions; }
}
```

Aquí tienes una clase Foo con métodos decorados de varias formas con el atributo Test:

```
class Foo
{
  [Test]
  public void Method1() { ... }

  [Test(20)]
  public void Method2() { ... }

  [Test(20, FailureMessage="Debugging Time!")]
  public void Method3() { ... }
}
```

Recuperar atributos en tiempo de ejecución

Hay dos formas estándar de recuperar atributos en tiempo de ejecución:

- Llamar a GetCustomAttributes en cualquier objeto Type o MemberInfo.
- Llamar a Attribute.GetCustomAttribute o Attribute.GetCustomAttributes.

Estos dos últimos métodos están sobrecargados para aceptar cualquier objeto de reflexión que corresponda a un destino de atributo válido (Type, Assembly, Module, MemberInfo o ParameterInfo).



También puedes llamar a GetCustomAttributesData() en un tipo o miembro para obtener información sobre el atributo. La diferencia entre esto y GetCustomAttributes() es que el primero te permite saber cómo se instanció el atributo: informa sobre la sobrecarga del constructor que se utilizó y el valor de cada argumento de constructor y parámetro con nombre. Esto es útil cuando deseas emitir código o IL para reconstruir el atributo en el mismo estado (consultar "Emitting Type Members" en la página 818).

Aquí te muestro cómo enumerar cada método en la clase Foo anterior que tiene un TestAttribute:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
{
  TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
    (mi, typeof (TestAttribute));

  if (att != null)
    Console.WriteLine ("Method {0} will be tested; reps={1}; msg={2}",
                       mi.Name, att.Repetitions, att.FailureMessage);
}
```

O también:

```
foreach (MethodInfo mi in typeof (Foo).GetTypeInfo().DeclaredMethods)
...
```

Aquí está la salida:

```
Method Method1 will be tested; reps=1; msg=
Method Method2 will be tested; reps=20; msg=
Method Method3 will be tested; reps=20; msg=Debugging Time!
```

Para completar la ilustración sobre cómo podríamos usar esto para escribir un sistema de pruebas unitarias, aquí tienes el mismo ejemplo expandido para que realmente llame a los métodos decorados con el atributo Test:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
{
  TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
    (mi, typeof (TestAttribute));


  if (att != null)
    for (int i = 0; i < att.Repetitions; i++)
      try
      {
        mi.Invoke (new Foo(), null);    // Call method with no arguments
      }
      catch (Exception ex)       // Wrap exception in att.FailureMessage
      {
        throw new Exception ("Error: " + att.FailureMessage, ex);
      }
}
```

Regresando a la reflexión de atributos, aquí tienes un ejemplo que enumera los atributos presentes en un tipo específico:

```
object[] atts = Attribute.GetCustomAttributes (typeof (Test));
foreach (object att in atts) Console.WriteLine (att);

[Serializable, Obsolete]
class Test
{
}
```

Y aquí está la salida:

```
System.ObsoleteAttribute
System.SerializableAttribute
```

Generación de código dinámico

El espacio de nombres System.Reflection.Emit contiene clases para crear metadatos e IL en tiempo de ejecución. Generar código dinámicamente es útil para ciertos tipos de tareas de programación. Un ejemplo es la API de expresiones regulares, que emite tipos de alto rendimiento ajustados a expresiones regulares específicas. Otro ejemplo es Entity Framework Core, que utiliza Reflection.Emit para generar clases proxy que permiten la carga diferida.

Generación de IL con DynamicMethod

La clase DynamicMethod es una herramienta ligera en el espacio de nombres System.Reflection.Emit para generar métodos sobre la marcha. A diferencia de TypeBuilder, no requiere que primero configures un ensamblado, un módulo y un tipo dinámicos en los que se contenga el método. Esto lo hace adecuado para tareas simples, así como para servir como una buena introducción a Reflection.Emit.



Un DynamicMethod y el IL asociado se recolectan como basura cuando ya no se hace referencia a ellos. Esto significa que puedes generar métodos dinámicos repetidamente sin llenar la memoria. (Para hacer lo mismo con ensamblados dinámicos, debes aplicar la marca AssemblyBuilderAccess.RunAndCollect al crear el ensamblado).

Aquí tienes un ejemplo simple de uso de DynamicMethod para crear un método que escribe "Hello world" en la consola:

```
public class Test
{
  static void Main()
  {
    var dynMeth = new DynamicMethod ("Foo", null, null, typeof (Test));
    ILGenerator gen = dynMeth.GetILGenerator();
    gen.EmitWriteLine ("Hello world");
    gen.Emit (OpCodes.Ret);
    dynMeth.Invoke (null, null);                    // Hello world
  }
}
```

La clase `OpCodes` tiene un campo de solo lectura estático para cada código IL. La mayoría de las funcionalidades se exponen a través de varios códigos de operación (opcodes), aunque `ILGenerator` también tiene métodos especializados para generar etiquetas, variables locales y manejo de excepciones.

Un método siempre termina con `OpCodes.Ret`, que significa "retornar", o alguna instrucción de ramificación/lanzamiento. El método `EmitWriteLine` en `ILGenerator` es un atajo para emitir

una serie de opcodes de nivel inferior. Obtendríamos el mismo resultado si reemplazamos la llamada a `EmitWriteLine` con esto:

```
MethodInfo writeLineStr = typeof (Console).GetMethod ("WriteLine",
                          new Type[] { typeof (string) });
gen.Emit (OpCodes.Ldstr, "Hello world");     // Load a string
gen.Emit (OpCodes.Call, writeLineStr);       // Call a method
```

Observa que pasamos `typeof(Test)` al constructor de `DynamicMethod`. Esto le da al método dinámico acceso a los métodos no públicos de ese tipo, lo que nos permite hacer esto:

```
public class Test
{
  static void Main()
  {
    var dynMeth = new DynamicMethod ("Foo", null, null, typeof (Test));
    ILGenerator gen = dynMeth.GetILGenerator();

    MethodInfo privateMethod = typeof(Test).GetMethod ("HelloWorld",
      BindingFlags.Static | BindingFlags.NonPublic);

    gen.Emit (OpCodes.Call, privateMethod);     // Call HelloWorld
    gen.Emit (OpCodes.Ret);

    dynMeth.Invoke (null, null);                // Hello world
  }

  static void HelloWorld()       // private method, yet we can call it
  {
    Console.WriteLine ("Hello world");
  }
}
```

Comprender el lenguaje IL requiere una considerable inversión de tiempo. En lugar de entender todos los opcodes, es mucho más fácil compilar un programa en C# y luego examinar, copiar y ajustar el IL. LINQPad muestra el IL de cualquier método o fragmento de código que escribas, y las herramientas de visualización de ensamblados como ILSpy son útiles para examinar ensamblados existentes.

La pila de evaluación (The Evaluation Stack)

El concepto central del IL es la pila de evaluación. Para llamar a un método con argumentos, primero se empujan ("cargan") los argumentos en la pila de evaluación y luego se llama al método. El método luego desapila los argumentos que necesita de la pila de evaluación. Esto lo demostramos previamente al llamar a `Console.WriteLine`. Aquí tienes un ejemplo similar con un número entero:

```
var dynMeth = new DynamicMethod ("Foo", null, null, typeof(void));
ILGenerator gen = dynMeth.GetILGenerator();
MethodInfo writeLineInt = typeof (Console).GetMethod ("WriteLine",
                                        new Type[] { typeof (int) });

// The Ldc* op-codes load numeric literals of various types and sizes.

gen.Emit (OpCodes.Ldc_I4, 123);        // Push a 4-byte integer onto stack
gen.Emit (OpCodes.Call, writeLineInt);

gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);           // 123
```

Para sumar dos números, primero se cargan cada número en la pila de evaluación y luego se llama a `Add`. El opcode Add desapila dos valores de la pila de evaluación y empuja el resultado de vuelta. El siguiente ejemplo suma 2 y 2, y luego escribe el resultado usando el método `writeLine` obtenido anteriormente:

```
gen.Emit (OpCodes.Ldc_I4, 2);           // Push a 4-byte integer, value=2
gen.Emit (OpCodes.Ldc_I4, 2);           // Push a 4-byte integer, value=2
gen.Emit (OpCodes.Add);                 // Add the result together
gen.Emit (OpCodes.Call, writeLineInt);
```

Para calcular 10 / 2 + 1, puedes hacer esto:

```
gen.Emit (OpCodes.Ldc_I4, 10);
gen.Emit (OpCodes.Ldc_I4, 2);
gen.Emit (OpCodes.Div);
gen.Emit (OpCodes.Ldc_I4, 1);
gen.Emit (OpCodes.Add);
gen.Emit (OpCodes.Call, writeLineInt);
```

o esto:

```
gen.Emit (OpCodes.Ldc_I4, 1);
gen.Emit (OpCodes.Ldc_I4, 10);
gen.Emit (OpCodes.Ldc_I4, 2);
gen.Emit (OpCodes.Div);
gen.Emit (OpCodes.Add);
gen.Emit (OpCodes.Call, writeLineInt);
```

Pasando argumentos a un método dinámico

Los opcodes Ldarg y Ldarg_XXX cargan un argumento pasado a un método en la pila de evaluación. Para devolver un valor, debes dejar exactamente un valor en la pila al finalizar. Para que esto funcione, debes especificar el tipo de retorno y los tipos de argumentos al llamar al constructor de DynamicMethod. El siguiente ejemplo crea un método dinámico que devuelve la suma de dos enteros:

```
DynamicMethod dynMeth = new DynamicMethod ("Foo",
  typeof (int),                              // Return type = int
  new[] { typeof (int), typeof (int) },      // Parameter types = int, int
  typeof (void));

ILGenerator gen = dynMeth.GetILGenerator();

gen.Emit (OpCodes.Ldarg_0);      // Push first arg onto eval stack
gen.Emit (OpCodes.Ldarg_1);      // Push second arg onto eval stack
gen.Emit (OpCodes.Add);          // Add them together (result on stack)
gen.Emit (OpCodes.Ret);          // Return with stack having 1 value

int result = (int) dynMeth.Invoke (null, new object[] { 3, 4 } );   // 7
```



Cuando finalices, la pila de evaluación debe tener exactamente 0 o 1 elemento (dependiendo de si el método devuelve un valor). Si no se cumple esta condición, el CLR se negará a ejecutar el método. Puedes eliminar un elemento de la pila sin procesarlo utilizando el opcode OpCodes.Pop.

En lugar de llamar a Invoke, puede ser más conveniente trabajar con un método dinámico como un delegado tipado. El método CreateDelegate logra precisamente eso. En nuestro caso, el delegado que necesitamos tiene dos parámetros enteros y devuelve un entero. Podemos utilizar el delegado Func<int, int, int> para este propósito. La última línea de nuestro ejemplo anterior se convierte en lo siguiente:

```
var func = (Func<int,int,int>) dynMeth.CreateDelegate
                                 (typeof (Func<int,int,int>));
int result = func (3, 4);  // 7
```



Un delegado también elimina la sobrecarga de la invocación de métodos dinámicos, lo que ahorra unos pocos microsegundos por llamada.

Enseñamos cómo pasar por referencia en "Generación de miembros de tipos" en la página 818.

Generación de variables locales

Puedes declarar una variable local llamando a DeclareLocal en un ILGenerator. Esto devuelve un objeto LocalBuilder, que puedes utilizar junto con opcodes como Ldloc (cargar una variable local) o Stloc (almacenar en una variable local). Ldloc empuja la pila de evaluación; Stloc la extrae. Por ejemplo, considera el siguiente código en C#:

```
int x = 6;
int y = 7;
x *= y;
Console.WriteLine (x);
```

El siguiente código genera el código anterior de forma dinámica:

```
var dynMeth = new DynamicMethod ("Test", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator();

LocalBuilder localX = gen.DeclareLocal (typeof (int));    // Declare x
LocalBuilder localY = gen.DeclareLocal (typeof (int));    // Declare y

gen.Emit (OpCodes.Ldc_I4, 6);        // Push literal 6 onto eval stack
gen.Emit (OpCodes.Stloc, localX);    // Store in localX
gen.Emit (OpCodes.Ldc_I4, 7);        // Push literal 7 onto eval stack
gen.Emit (OpCodes.Stloc, localY);    // Store in localY

gen.Emit (OpCodes.Ldloc, localX);    // Push localX onto eval stack
gen.Emit (OpCodes.Ldloc, localY);    // Push localY onto eval stack
gen.Emit (OpCodes.Mul);              // Multiply values together
gen.Emit (OpCodes.Stloc, localX);    // Store the result to localX

gen.EmitWriteLine (localX);          // Write the value of localX
gen.Emit (OpCodes.Ret);

dynMeth.Invoke (null, null);         // 42
```

Ramas (Branching)

En IL, no hay bucles while, do y for; todo se hace con etiquetas (labels) y las instrucciones equivalentes a goto y goto condicional. Estas son las opcodes de ramificación, como Br (ramificar incondicionalmente), Brtrue (ramificar si el valor en la pila de evaluación es verdadero) y Blt (ramificar si el primer valor es menor que el segundo valor).

Para establecer un objetivo de ramificación, primero llama a DefineLabel (esto devuelve un objeto Label) y luego llama a MarkLabel en el lugar donde deseas anclar la etiqueta. Por ejemplo, considera el siguiente código en C#:

```
int x = 5;
while (x <= 10) Console.WriteLine (x++);
```

Podemos generar esto de la siguiente manera:

```
ILGenerator gen = ...

Label startLoop = gen.DefineLabel();                  // Declare labels
Label endLoop = gen.DefineLabel();

LocalBuilder x = gen.DeclareLocal (typeof (int));     // int x
gen.Emit (OpCodes.Ldc_I4, 5);                         //
gen.Emit (OpCodes.Stloc, x);                          // x = 5
gen.MarkLabel (startLoop);
  gen.Emit (OpCodes.Ldc_I4, 10);              // Load 10 onto eval stack
  gen.Emit (OpCodes.Ldloc, x);                // Load x onto eval stack
```

```
    gen.Emit (OpCodes.Blt, endLoop);            // if (x > 10) goto endLoop

    gen.EmitWriteLine (x);                      // Console.WriteLine (x)

    gen.Emit (OpCodes.Ldloc, x);                // Load x onto eval stack
    gen.Emit (OpCodes.Ldc_I4, 1);               // Load 1 onto the stack
    gen.Emit (OpCodes.Add);                     // Add them together
    gen.Emit (OpCodes.Stloc, x);                // Save result back to x

    gen.Emit (OpCodes.Br, startLoop);           // return to start of loop
  gen.MarkLabel (endLoop);

  gen.Emit (OpCodes.Ret);
```

Crear objetos e invocar métodos de instancia

La instrucción IL equivalente a `new` es la opcode `Newobj`. Esta opcode toma un constructor y carga el objeto construido en la pila de evaluación. Por ejemplo, el siguiente código construye un objeto `StringBuilder`:

```
var dynMeth = new DynamicMethod ("Test", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator();

ConstructorInfo ci = typeof (StringBuilder).GetConstructor (new Type[0]);
gen.Emit (OpCodes.Newobj, ci);
```

Después de cargar un objeto en la pila de evaluación, puedes utilizar las opcodes `Call` o `Callvirt` para invocar los métodos de instancia del objeto. Siguiendo con este ejemplo, consultaremos la propiedad `MaxCapacity` de `StringBuilder` llamando al accessor `get` de la propiedad y luego escribiremos el resultado:

```
gen.Emit (OpCodes.Callvirt, typeof (StringBuilder)
                           .GetProperty ("MaxCapacity").GetGetMethod());

gen.Emit (OpCodes.Call, typeof (Console).GetMethod ("WriteLine",
                                        new[] { typeof (int) } ));
gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);              // 2147483647
```

Para emular la semántica de llamada de C#:

- Utiliza `Call` para invocar métodos estáticos y métodos de instancia de tipos de valor.
- Utiliza `Callvirt` para invocar métodos de instancia de tipos de referencia (ya sean declarados como virtuales o no).

En nuestro ejemplo, utilizamos `Callvirt` en la instancia de `StringBuilder`, aunque `MaxCapacity` no es virtual. Esto no causa un error, simplemente realiza una llamada no virtual. Siempre invocar métodos de instancia de tipos de referencia con `Callvirt` evita el riesgo de invocar accidentalmente un método virtual con `Call` (El riesgo es real, ya que el autor del método de destino puede cambiar su declaración en el futuro).

Además, `Callvirt` tiene la ventaja de verificar que el receptor no sea nulo antes de la llamada.



Invocar un método virtual con Call omite la semántica de llamada virtual y llama directamente a ese método. Esto rara vez es deseable y, en efecto, viola la seguridad de tipos.

En el siguiente ejemplo, construimos un StringBuilder pasando dos argumentos, agregamos ", mundo!" al StringBuilder y luego llamamos a ToString en él:

```
// We will call:   new StringBuilder ("Hello", 1000)

ConstructorInfo ci = typeof (StringBuilder).GetConstructor (
                     new[] { typeof (string), typeof (int) } );

gen.Emit (OpCodes.Ldstr, "Hello");   // Load a string onto the eval stack
gen.Emit (OpCodes.Ldc_I4, 1000);     // Load an int onto the eval stack
gen.Emit (OpCodes.Newobj, ci);       // Construct the StringBuilder

Type[] strT = { typeof (string) };
gen.Emit (OpCodes.Ldstr, ", world!");
gen.Emit (OpCodes.Call, typeof (StringBuilder).GetMethod ("Append", strT));
gen.Emit (OpCodes.Callvirt, typeof (object).GetMethod ("ToString"));
gen.Emit (OpCodes.Call, typeof (Console).GetMethod ("WriteLine", strT));
gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);        // Hello, world!
```

Por diversión, llamamos a GetMethod en typeof(object) y luego usamos Callvirt para realizar una llamada de método virtual a ToString. Podríamos haber obtenido el mismo resultado llamando a ToString en el tipo StringBuilder en sí mismo:

```
gen.Emit (OpCodes.Callvirt, typeof (StringBuilder).GetMethod ("ToString",
                                                    new Type[0] ));
```

(El arreglo vacío de tipos es necesario al llamar a GetMethod porque StringBuilder sobrecarga ToString con otra firma).



Si hubiéramos llamado al método ToString de object de manera no virtual como esto:

```
gen.Emit (OpCodes.Call,
          typeof (object).GetMethod ("ToString"));
```

el resultado habría sido System.Text.StringBuilder. En otras palabras, habríamos evitado la anulación de ToString de StringBuilder y habríamos llamado directamente a la versión de object.

Control de excepciones

ILGenerator proporciona métodos dedicados para el control de excepciones. Por lo tanto, la traducción para este código C# sería:

```
try                               { throw new NotSupportedException(); }
catch (NotSupportedException ex)  { Console.WriteLine (ex.Message);    }
finally                           { Console.WriteLine ("Finally");     }
```

esto:

```
MethodInfo getMessageProp = typeof (NotSupportedException)
                            .GetProperty ("Message").GetGetMethod();

MethodInfo writeLineString = typeof (Console).GetMethod ("WriteLine",
                                          new[] { typeof (object) } );
gen.BeginExceptionBlock();
  ConstructorInfo ci = typeof (NotSupportedException).GetConstructor (
                                                      new Type[0] );
  gen.Emit (OpCodes.Newobj, ci);
  gen.Emit (OpCodes.Throw);
gen.BeginCatchBlock (typeof (NotSupportedException));
  gen.Emit (OpCodes.Callvirt, getMessageProp);
  gen.Emit (OpCodes.Call, writeLineString);
gen.BeginFinallyBlock();
  gen.EmitWriteLine ("Finally");
gen.EndExceptionBlock();
```

Así como en C#, puedes incluir múltiples bloques catch. Para relanzar la misma excepción, emite el opcode Rethrow.



ILGenerator proporciona un método auxiliar llamado ThrowException. Sin embargo, este método contiene un error que impide su uso con DynamicMethod. Solo funciona con un MethodBuilder (ver la siguiente sección).

Emisión de ensamblados y tipos

Aunque DynamicMethod es conveniente, solo puede generar métodos. Si necesitas emitir cualquier otra construcción, o un tipo completo, debes utilizar la API completa "pesada". Esto significa construir dinámicamente un ensamblado y un módulo.

El ensamblado no necesita tener presencia en disco (de hecho, no puede tenerla, ya que .NET 5+ y .NET Core no permiten guardar ensamblados generados en disco).

Supongamos que queremos construir dinámicamente un tipo. Dado que un tipo debe residir en un módulo dentro de un ensamblado, primero debemos crear el ensamblado y el módulo antes de poder crear el tipo. Esto es tarea de los tipos AssemblyBuilder y ModuleBuilder:

```
AssemblyName aname = new AssemblyName ("MyDynamicAssembly");

AssemblyBuilder assemBuilder =
  AssemblyBuilder.DefineDynamicAssembly (aname, AssemblyBuilderAccess.Run);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule ("DynModule");
```



No puedes agregar un tipo a un ensamblado existente, ya que un ensamblado es inmutable después de haber sido creado.

Los ensamblados dinámicos no son recolectados por el recolector de basura y permanecen en memoria hasta que finaliza el proceso, a menos que especifiques AssemblyBuilderAccess.RunAndCollect al definir el ensamblado. Se aplican varias restricciones a los ensamblados recolectables (ver http://albahari.com/dynamiccollect).

Una vez que tenemos un módulo en el cual el tipo puede residir, podemos usar TypeBuilder para crear el tipo. A continuación, se define una clase llamada Widget:

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
```

El enum TypeAttributes admite los modificadores de tipo CLR que se ven al desensamblar un tipo con ildasm. Además de los indicadores de visibilidad de miembros, esto incluye modificadores de tipo como Abstract y Sealed, e Interface para definir una interfaz en .NET. También incluye Serializable, que es equivalente a aplicar el atributo [Serializable] en C#, y Explicit, que es equivalente a aplicar [StructLayout(LayoutKind.Explicit)]. Describimos cómo aplicar otros tipos de atributos más adelante en este capítulo, en "Adjuntar atributos" en la página 823.



El método DefineType también acepta un tipo base opcional:

- Para definir una estructura, especifica un tipo base de System.ValueType.
- Para definir un delegado, especifica un tipo base de System.MulticastDelegate.
- Para implementar una interfaz, utiliza el constructor que acepta una matriz de tipos de interfaz.
- Para definir una interfaz, especifica TypeAttributes.Interface | TypeAttributes.Abstract.

Definir un tipo de delegado requiere algunos pasos adicionales. En su blog, Joel Pobar demuestra cómo hacerlo en su artículo titulado "Creating delegate types via Reflection.Emit".

Ahora podemos crear miembros dentro del tipo:

```
MethodBuilder methBuilder = tb.DefineMethod ("SayHello",
                                              MethodAttributes.Public,
                                              null, null);
ILGenerator gen = methBuilder.GetILGenerator();
gen.EmitWriteLine ("Hello world");
gen.Emit (OpCodes.Ret);
```

Ahora estamos listos para crear el tipo, lo cual finaliza su definición:

```
Type t = tb.CreateType();
```

Después de crear el tipo, podemos usar la reflexión ordinaria para inspeccionarlo y realizar enlaces tardíos (late binding):

```
object o = Activator.CreateInstance (t);
t.GetMethod ("SayHello").Invoke (o, null);        // Hello world
```

El Modelo de Objetos Reflection.Emit

La Figura 18-2 ilustra los tipos esenciales en System.Reflection.Emit. Cada tipo describe una construcción del CLR y se basa en un equivalente en el espacio de nombres System.Reflection. Esto te permite utilizar construcciones emitidas en lugar de construcciones normales al crear un tipo. Por ejemplo, anteriormente llamamos a Console.WriteLine de la siguiente manera:

```
MethodInfo writeLine = typeof(Console).GetMethod ("WriteLine",
                                    new Type[] { typeof (string) });
gen.Emit (OpCodes.Call, writeLine);
```

Podríamos llamar igualmente a un método generado dinámicamente llamando a gen.Emit con un MethodBuilder en lugar de un MethodInfo. Esto es esencial, de lo contrario no podrías escribir un método dinámico que llame a otro en el mismo tipo.

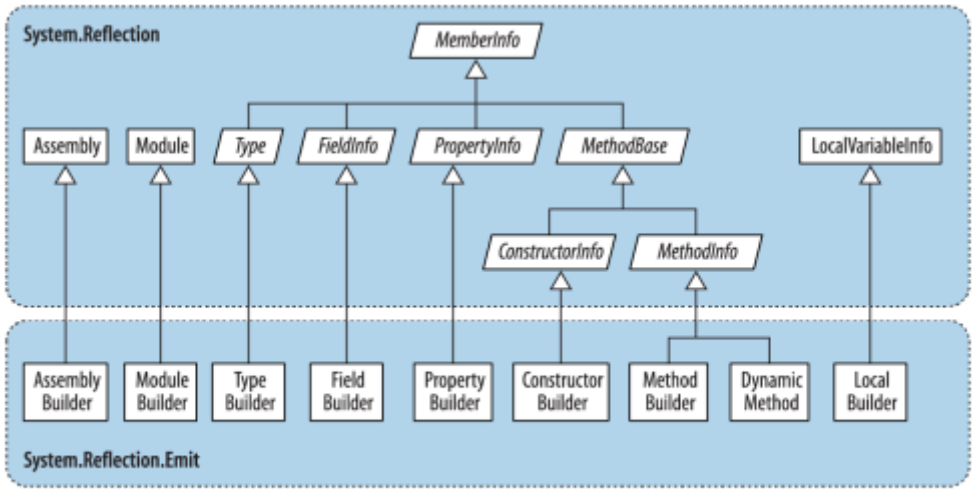


Figure 18-2. System.Reflection.Emit

Recuerda que debes llamar a CreateType en un TypeBuilder cuando hayas terminado de poblarlo. Llamar a CreateType sella el TypeBuilder y todos sus miembros, por lo que no se puede agregar ni modificar nada más, y te devuelve un verdadero Type que puedes instanciar.

Antes de llamar a CreateType, el TypeBuilder y sus miembros están en un estado "no creado". Hay restricciones significativas sobre lo que puedes hacer con construcciones no creadas. En particular, no puedes llamar a ninguno de los miembros que devuelven objetos MemberInfo, como GetMembers, GetMethod o GetProperty, ya que todos arrojan una excepción. Si quieres hacer referencia a miembros de un tipo no creado, debes usar las emisiones originales:

```
TypeBuilder tb = ...

MethodBuilder method1 = tb.DefineMethod ("Method1", ...);
MethodBuilder method2 = tb.DefineMethod ("Method2", ...);

ILGenerator gen1 = method1.GetILGenerator();

// Suppose we want method1 to call method2:

gen1.Emit (OpCodes.Call, method2);                    // Right
gen1.Emit (OpCodes.Call, tb.GetMethod ("Method2"));   // Wrong
```

Después de llamar a CreateType, puedes reflejar y activar no solo el Type devuelto, sino también el objeto original TypeBuilder. De hecho, el TypeBuilder se convierte en un proxy para el verdadero Type. Verás por qué esta característica es importante en "Objetivos de emisión incómodos" en la página 825.

Emisión de miembros de tipo

Todos los ejemplos en esta sección asumen que se ha instanciado un TypeBuilder, tb, de la siguiente manera:

```
AssemblyName aname = new AssemblyName ("MyEmissions");

AssemblyBuilder assemBuilder = AssemblyBuilder.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.Run);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule ("MainModule");

TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
```

Emisión de métodos

Puedes especificar un tipo de retorno y los tipos de parámetros al llamar a DefineMethod, de la misma manera que al instanciar un DynamicMethod. Por ejemplo, el siguiente método:

```
public static double SquareRoot (double value) => Math.Sqrt (value);
```

puede generarse así:

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Static | MethodAttributes.Public,
  CallingConventions.Standard,
  typeof (double),                    // Return type
  new[]  { typeof (double) } );       // Parameter types

mb.DefineParameter (1, ParameterAttributes.None, "value");  // Assign name

ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);                                 // Load 1st arg
gen.Emit (OpCodes.Call, typeof(Math).GetMethod ("Sqrt"));
gen.Emit (OpCodes.Ret);

Type realType = tb.CreateType();
double x = (double) tb.GetMethod ("SquareRoot").Invoke (null,
                                          new object[] { 10.0 });
Console.WriteLine (x);   // 3.16227766016838
```

Llamar a DefineParameter es opcional y se hace típicamente para asignar un nombre al parámetro. El número 1 se refiere al primer parámetro (0 se refiere al valor de retorno). Si llamas a DefineParameter, el parámetro se nombra implícitamente como __p1, __p2, y así sucesivamente. Asignar nombres tiene sentido si vas a escribir el ensamblado en disco; hace que tus métodos sean amigables para los consumidores.



El método DefineParameter devuelve un objeto ParameterBuilder en el que puedes llamar a SetCustomAttribute para adjuntar atributos (ver "Adjuntar atributos" en la página 823).

Para emitir parámetros pasados por referencia, como en el siguiente método en C#:

```
public static void SquareRoot (ref double value)
  => value = Math.Sqrt (value);
```

llama a MakeByRefType en el tipo de parámetro(s):

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Static | MethodAttributes.Public,
  CallingConventions.Standard,
  null,
  new Type[] { typeof (double).MakeByRefType() } );

mb.DefineParameter (1, ParameterAttributes.None, "value");

ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ldind_R8);
gen.Emit (OpCodes.Call, typeof (Math).GetMethod ("Sqrt"));
gen.Emit (OpCodes.Stind_R8);
gen.Emit (OpCodes.Ret);

Type realType = tb.CreateType();
object[] args = { 10.0 };
tb.GetMethod ("SquareRoot").Invoke (null, args);
Console.WriteLine (args[0]);                    // 3.16227766016838
```

Los opcodes aquí fueron copiados de un método en C# desensamblado. Observa la diferencia en la semántica para acceder a parámetros pasados por referencia: Ldind y Stind significan "cargar indirectamente" y "almacenar indirectamente", respectivamente. El sufijo R8 se refiere a un número de punto flotante de ocho bytes.

El proceso para emitir parámetros de salida es idéntico, excepto que llamas a DefineParameter de la siguiente manera:

```
mb.DefineParameter (1, ParameterAttributes.Out, "value");
```

Generando métodos de instancia

Para generar un método de instancia, especifica MethodAttributes.Instance al llamar a DefineMethod:

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Instance | MethodAttributes.Public
  ...
```

Con los métodos de instancia, el argumento cero es implícitamente el objeto "this"; los argumentos restantes comienzan en 1. Por lo tanto, Ldarg_0 carga el objeto "this" en la pila de evaluación; Ldarg_1 carga el primer argumento real del método.

Sobrescribir métodos

Sobrescribir un método virtual en una clase base es fácil: simplemente define un método con el mismo nombre, firma y tipo de retorno, especificando MethodAttributes.Virtual al llamar a DefineMethod. Lo mismo se aplica al implementar métodos de interfaces.

TypeBuilder también expone un método llamado DefineMethodOverride que sobrescribe un método con un nombre diferente. Esto tiene sentido solo con la implementación explícita de una interfaz; en otros escenarios, usa DefineMethod.

HideBySig

Si estás heredando de otra clase, casi siempre vale la pena especificar MethodAttributes.HideBySig al definir métodos. HideBySig asegura que se apliquen las semánticas de ocultamiento de métodos estilo C#, lo que significa que un método base se oculta solo si un subtipo define un método con la misma firma. Sin HideBySig, el ocultamiento de métodos considera solo el nombre, por lo que Foo(string) en el subtipo ocultará a Foo() en el tipo base, lo cual generalmente no es deseable.

Emitir campos y propiedades

Para crear un campo, llamas a DefineField en un TypeBuilder, especificando el nombre del campo deseado, el tipo y la visibilidad. Lo siguiente crea un campo privado de tipo entero llamado "length":

```
FieldBuilder field = tb.DefineField ("length", typeof (int),
                                     FieldAttributes.Private);
```

Crear una propiedad o un indexador requiere algunos pasos adicionales. Primero, llama a DefineProperty en un TypeBuilder, proporcionándole el nombre y el tipo de la propiedad:

```
PropertyBuilder prop = tb.DefineProperty (
                         "Text",                    // Name of property
                         PropertyAttributes.None,
                         typeof (string),           // Property type
                         new Type[0]                // Indexer types
                       );
```

(Los indexadores requieren un argumento final que sea una matriz de tipos de indexador). Observa que no hemos especificado la visibilidad de la propiedad, esto se hace individualmente en los métodos de acceso.

El siguiente paso es escribir los métodos get y set. Por convención, sus nombres se prefijan con "get_" o "set_". Luego los adjuntas a la propiedad llamando a SetGetMethod y SetSetMethod en el PropertyBuilder.

Para dar un ejemplo completo, tomemos la siguiente declaración de campo y propiedad:

```
string _text;
public string Text
{
  get          => _text;
  internal set => _text = value;
}
```

y generémoslo dinámicamente:

```
FieldBuilder field = tb.DefineField ("_text", typeof (string),
                                     FieldAttributes.Private);
PropertyBuilder prop = tb.DefineProperty (
                         "Text",                      // Name of property
                         PropertyAttributes.None,
                         typeof (string),             // Property type
                         new Type[0]);                // Indexer types

MethodBuilder getter = tb.DefineMethod (
  "get_Text",                                         // Method name
  MethodAttributes.Public | MethodAttributes.SpecialName,
  typeof (string),                                    // Return type
  new Type[0]);                                       // Parameter types

ILGenerator getGen = getter.GetILGenerator();
getGen.Emit (OpCodes.Ldarg_0);        // Load "this" onto eval stack
getGen.Emit (OpCodes.Ldfld, field);   // Load field value onto eval stack
getGen.Emit (OpCodes.Ret);            // Return

MethodBuilder setter = tb.DefineMethod (
  "set_Text",
  MethodAttributes.Assembly | MethodAttributes.SpecialName,
  null,                                                 // Return type
  new Type[] { typeof (string) } );                     // Parameter types

ILGenerator setGen = setter.GetILGenerator();
setGen.Emit (OpCodes.Ldarg_0);        // Load "this" onto eval stack
setGen.Emit (OpCodes.Ldarg_1);        // Load 2nd arg, i.e., value
setGen.Emit (OpCodes.Stfld, field);   // Store value into field
setGen.Emit (OpCodes.Ret);            // return

prop.SetGetMethod (getter);           // Link the get method and property
prop.SetSetMethod (setter);           // Link the set method and property
```

Podemos probar la propiedad de la siguiente manera:

```
Type t = tb.CreateType();
object o = Activator.CreateInstance (t);
t.GetProperty ("Text").SetValue (o, "Good emissions!", new object[0]);
string text = (string) t.GetProperty ("Text").GetValue (o, null);

Console.WriteLine (text);             // Good emissions!
```

Observa que al definir los atributos de acceso de los métodos, incluimos SpecialName. Esto instruye a los compiladores a prohibir la vinculación directa a estos métodos cuando se hace referencia estáticamente al ensamblado. También asegura que los accesores sean tratados adecuadamente por las herramientas de reflexión y la función IntelliSense de Visual Studio.



También puedes emitir eventos de manera similar, llamando a DefineEvent en un TypeBuilder. Luego, debes escribir los métodos de acceso de eventos de forma explícita y adjuntarlos al EventBuilder llamando a SetAddOnMethod y SetRemoveOnMethod.

Emitir Constructores

Puedes definir tus propios constructores llamando a DefineConstructor en un TypeBuilder. No estás obligado a hacerlo, ya que se proporciona automáticamente un constructor predeterminado sin parámetros si no lo haces. El constructor predeterminado llama al constructor de la clase base si se realiza una subclase, al igual que en C#. Definir uno o más constructores desplaza este constructor predeterminado.

Si necesitas inicializar campos, el constructor es un buen lugar para hacerlo. De hecho, es el único lugar: los inicializadores de campos de C# no tienen soporte especial en el CLR, simplemente son un atajo sintáctico para asignar valores a los campos en el constructor.

Entonces, para reproducir esto:

```
class Widget
{
  int _capacity = 4000;
}
```

debes definir un constructor de la siguiente manera:

```
FieldBuilder field = tb.DefineField ("_capacity", typeof (int),
                                     FieldAttributes.Private);
ConstructorBuilder c = tb.DefineConstructor (
  MethodAttributes.Public,
  CallingConventions.Standard,
  new Type[0]);                  // Constructor parameters

ILGenerator gen = c.GetILGenerator();

gen.Emit (OpCodes.Ldarg_0);            // Load "this" onto eval stack
gen.Emit (OpCodes.Ldc_I4, 4000);       // Load 4000 onto eval stack
gen.Emit (OpCodes.Stfld, field);       // Store it to our field
gen.Emit (OpCodes.Ret);
```

Llamando a constructores base

Si estás subclasificando otro tipo, el constructor que acabamos de escribir evita el constructor de la clase base. Esto es diferente a C#, donde el constructor de la clase base siempre se llama, ya sea directa o indirectamente. Por ejemplo, dado el siguiente código:

```
class A     { public A() { Console.Write ("A"); } }
class B : A { public B() {} }
```

el compilador, en efecto, traducirá la segunda línea en esto:

```
class B : A { public B() : base() {} }
```

Este no es el caso al generar IL: debes llamar explícitamente al constructor base si deseas que se ejecute (lo cual casi siempre es necesario). Suponiendo que la clase base se llama A, aquí te muestro cómo hacerlo:

```
gen.Emit (OpCodes.Ldarg_0);
ConstructorInfo baseConstr = typeof (A).GetConstructor (new Type[0]);
gen.Emit (OpCodes.Call, baseConstr);
```

Llamar a constructores con argumentos es similar a llamar a métodos.

Adjuntar atributos

Puedes adjuntar atributos personalizados a una construcción dinámica llamando a SetCustomAttribute con un CustomAttributeBuilder. Por ejemplo, supongamos que queremos adjuntar la siguiente declaración de atributo a un campo o propiedad:

```
[XmlElement ("FirstName", Namespace="http://test/", Order=3)]
```

Esto se basa en el constructor XmlElementAttribute que acepta una cadena única. Para utilizar CustomAttributeBuilder, debemos recuperar este constructor, así como las dos propiedades adicionales que deseamos establecer (Namespace y Order):

```
Type attType = typeof (XmlElementAttribute);

ConstructorInfo attConstructor = attType.GetConstructor (
  new Type[] { typeof (string) } );

var att = new CustomAttributeBuilder (
  attConstructor,                       // Constructor
  new object[] { "FirstName" },         // Constructor arguments
  new PropertyInfo[]
  {
    attType.GetProperty ("Namespace"),  // Properties
    attType.GetProperty ("Order")
  },
  new object[] { "http://test/", 3 }    // Property values
);

myFieldBuilder.SetCustomAttribute (att);
// or propBuilder.SetCustomAttribute (att);
// or typeBuilder.SetCustomAttribute (att);  etc
```

Emitiendo métodos y tipos genéricos

Todos los ejemplos en esta sección asumen que modBuilder se ha instanciado de la siguiente manera:

```
AssemblyName aname = new AssemblyName ("MyEmissions");

AssemblyBuilder assemBuilder = AssemblyBuilder.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.Run);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule ("MainModule");
```

Definición de métodos genéricos (Defining Generic Methods)

Para definir un método genérico, sigue estos pasos:

1. Llama a DefineGenericParameters en un MethodBuilder para obtener una matriz de objetos GenericTypeParameterBuilder.

2. Llama a SetSignature en un MethodBuilder usando estos parámetros de tipo genérico.

3. Opcionalmente, nombra los parámetros como lo harías normalmente.

Por ejemplo, el siguiente método genérico:

```
public static T Echo<T> (T value)
{
  return value;
}
```

puede ser emitido de la siguiente manera:

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);

MethodBuilder mb = tb.DefineMethod ("Echo", MethodAttributes.Public |
                                           MethodAttributes.Static);
GenericTypeParameterBuilder[] genericParams
  = mb.DefineGenericParameters ("T");

mb.SetSignature (genericParams[0],     // Return type
                 null, null,
                 genericParams,        // Parameter types
                 null, null);

mb.DefineParameter (1, ParameterAttributes.None, "value");   // Optional

ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ret);
```

El método DefineGenericParameters acepta cualquier número de argumentos de tipo cadena, que corresponden a los nombres deseados de los tipos genéricos. En este ejemplo, solo necesitamos un tipo genérico llamado T. GenericTypeParameterBuilder se basa en System.Type, por lo que puedes usarlo en lugar de un TypeBuilder al emitir opcodes.

GenericTypeParameterBuilder también te permite especificar una restricción de tipo base:

```
genericParams[0].SetBaseTypeConstraint (typeof (Foo));
```

y restricciones de interfaz:

```
genericParams[0].SetInterfaceConstraints (typeof (IComparable));
```

Para replicar esto, simplemente sigue los pasos mencionados y ajusta los nombres y las restricciones según tus necesidades:

```
public static T Echo<T> (T value) where T : IComparable<T>
```

Escribirías lo siguiente:

```
genericParams[0].SetInterfaceConstraints (
  typeof (IComparable<>).MakeGenericType (genericParams[0]) );
```

Para otros tipos de restricciones, utiliza SetGenericParameterAttributes. Esto acepta un miembro del enum GenericParameterAttributes, que incluye los siguientes valores:

```
DefaultConstructorConstraint
NotNullableValueTypeConstraint
ReferenceTypeConstraint
Covariant
Contravariant
```

Los dos últimos son equivalentes a aplicar los modificadores out e in a los parámetros de tipo.

Definir Tipos Genéricos

Puedes definir tipos genéricos de manera similar. La diferencia es que llamas a DefineGenericParameters en el TypeBuilder en lugar del MethodBuilder. Entonces, para reproducir esto:

```
public class Widget<T>
{
  public T Value;
}
```

harías lo siguiente:

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);

GenericTypeParameterBuilder[] genericParams
  = tb.DefineGenericParameters ("T");

tb.DefineField ("Value", genericParams[0], FieldAttributes.Public);
```

Se pueden agregar restricciones genéricas, al igual que en un método.

Destinos de emisión incómodos (Awkward Emission Targets)

Todos los ejemplos en esta sección asumen que se ha instanciado un modBuilder como en las secciones anteriores.

Genéricos cerrados no creados

Supongamos que quieres emitir un método que utiliza un tipo genérico cerrado:

```
public class Widget
{
  public static void Test() { var list = new List<int>(); }
}
```

El proceso es bastante sencillo:

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);

MethodBuilder mb = tb.DefineMethod ("Test", MethodAttributes.Public |
                                            MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator();

Type variableType = typeof (List<int>);

ConstructorInfo ci = variableType.GetConstructor (new Type[0]);

LocalBuilder listVar = gen.DeclareLocal (variableType);
gen.Emit (OpCodes.Newobj, ci);
gen.Emit (OpCodes.Stloc, listVar);
gen.Emit (OpCodes.Ret);
```

Ahora supongamos que, en lugar de una lista de enteros, queremos una lista de widgets:

```
public class Widget
{
  public static void Test() { var list = new List<Widget>(); }
}
```

En teoría, esta es una modificación sencilla; simplemente reemplazamos esta línea:

```
Type variableType = typeof (List<int>);
```

con esta:

```
Type variableType = typeof (List<>).MakeGenericType (tb);
```

Desafortunadamente, esto provoca que se lance una NotSupportedException al llamar a GetConstructor. El problema radica en que no se puede llamar a GetConstructor en un tipo genérico cerrado con un TypeBuilder no creado. Lo mismo ocurre con GetField y GetMethod.

La solución es poco intuitiva. TypeBuilder proporciona tres métodos estáticos:

```
public static ConstructorInfo GetConstructor (Type, ConstructorInfo);
public static FieldInfo       GetField       (Type, FieldInfo);
public static MethodInfo      GetMethod      (Type, MethodInfo);
```

Aunque no parezca, estos métodos existen específicamente para obtener miembros de tipos genéricos cerrados con TypeBuilders no creados. El primer parámetro es el tipo genérico cerrado y el segundo parámetro es el miembro que deseas obtener del tipo genérico no ligado. Aquí tienes la versión corregida de nuestro ejemplo:

```
MethodBuilder mb = tb.DefineMethod ("Test", MethodAttributes.Public |
                                            MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator();

Type variableType = typeof (List<>).MakeGenericType (tb);

ConstructorInfo unbound = typeof (List<>).GetConstructor (new Type[0]);
ConstructorInfo ci = TypeBuilder.GetConstructor (variableType, unbound);

LocalBuilder listVar = gen.DeclareLocal (variableType);
gen.Emit (OpCodes.Newobj, ci);

gen.Emit (OpCodes.Stloc, listVar);
gen.Emit (OpCodes.Ret);
```

Dependencias circulares

Supongamos que deseas construir dos tipos que se referencian entre sí, como estos:

```
class A { public B Bee; }
class B { public A Aye; }
```

Puedes generar esto dinámicamente de la siguiente manera:

```
var publicAtt = FieldAttributes.Public;

TypeBuilder aBuilder = modBuilder.DefineType ("A");
TypeBuilder bBuilder = modBuilder.DefineType ("B");

FieldBuilder bee = aBuilder.DefineField ("Bee", bBuilder, publicAtt);
FieldBuilder aye = bBuilder.DefineField ("Aye", aBuilder, publicAtt);

Type realA = aBuilder.CreateType();
Type realB = bBuilder.CreateType();
```

Observa que no llamamos a CreateType en aBuilder o bBuilder hasta que hayamos populado ambos objetos. El principio es el siguiente: primero conecta todo y luego llama a CreateType en cada TypeBuilder.

Curiosamente, el tipo realA es válido pero disfuncional hasta que llames a CreateType en bBuilder. (Si comenzaras a usar aBuilder antes de esto, se lanzaría una excepción cuando intentes acceder al campo Bee).

Podrías preguntarte cómo bBuilder sabe cómo "arreglar" realA después de crear realB. La respuesta es que no lo sabe: realA puede arreglarse a sí mismo la próxima vez que se use. Esto es posible porque después de llamar a CreateType, un TypeBuilder se convierte en un proxy para el tipo real en tiempo de ejecución. Por lo tanto, realA, con sus referencias a bBuilder, puede obtener fácilmente los metadatos que necesita para la actualización.

Este sistema funciona cuando el TypeBuilder solo requiere información simple del tipo no construido, información que se puede predeterminar, como referencias de tipo, miembro y objeto. Al crear realA, el TypeBuilder no necesita saber, por ejemplo, cuántos bytes ocupará realB en la memoria en última instancia. ¡Y es mejor así porque realB aún no ha sido creado! Pero ahora imagina que realB es una estructura (struct). El tamaño final de realB se convierte en información crítica para crear realA.

Si la relación es no cíclica, por ejemplo:

```
struct A { public B Bee; }
struct B {               }
```

Puedes resolver esto creando primero la estructura B y luego la estructura A. Pero considera esto:

```
struct A { public B Bee; }
struct B { public A Aye; }
```

No intentaremos generar esto porque no tiene sentido que dos estructuras (struct) se contengan mutuamente (C# genera un error en tiempo de compilación si lo intentas). Pero la siguiente variación es legal y útil:

```
public struct S<T> { ... }    // S can be empty and this demo will work.

class A { S<B> Bee; }
class B { S<A> Aye; }
```

Al crear A, un TypeBuilder ahora necesita conocer el tamaño de memoria de B, y viceversa. Para ilustrarlo, asumamos que la estructura S está definida estáticamente. Aquí está el código para emitir las clases A y B:

```
var pub = FieldAttributes.Public;

TypeBuilder aBuilder = modBuilder.DefineType ("A");
TypeBuilder bBuilder = modBuilder.DefineType ("B");

aBuilder.DefineField ("Bee", typeof(S<>).MakeGenericType (bBuilder), pub);
bBuilder.DefineField ("Aye", typeof(S<>).MakeGenericType (aBuilder), pub);

Type realA = aBuilder.CreateType();    // Error: cannot load type B
Type realB = bBuilder.CreateType();
```

Ahora CreateType lanza una excepción de tipo TypeLoadException sin importar el orden en el que lo hagas:

- Si llamas primero aBuilder.CreateType, muestra "no se puede cargar el tipo B".
- Si llamas primero a bBuilder.CreateType, muestra "no se puede cargar el tipo A".

Para resolver esto, debes permitir que el TypeBuilder cree realB en parte durante la creación de realA. Esto se logra manejando el evento TypeResolve en la clase AppDomain justo antes de llamar a CreateType. Entonces, en nuestro ejemplo, reemplazamos las dos últimas líneas con esto:

```
TypeBuilder[] uncreatedTypes = { aBuilder, bBuilder };

ResolveEventHandler handler = delegate (object o, ResolveEventArgs args)
{
  var type = uncreatedTypes.FirstOrDefault (t => t.FullName == args.Name);
  return type == null ? null : type.CreateType().Assembly;
};

AppDomain.CurrentDomain.TypeResolve += handler;

Type realA = aBuilder.CreateType();
Type realB = bBuilder.CreateType();

AppDomain.CurrentDomain.TypeResolve -= handler;
```

El evento TypeResolve se activa durante la llamada a aBuilder.CreateType, en el punto en el que necesita que llames a CreateType en bBuilder.



Manejar el evento TypeResolve, como se muestra en este ejemplo, también es necesario cuando se define un tipo anidado y los tipos anidado y principal se refieren mutuamente.

Análisis de IL (Parsing IL)

Puedes obtener información sobre el contenido de un método existente llamando a GetMethodBody en un objeto MethodBase. Esto devuelve un objeto MethodBody que tiene propiedades para inspeccionar las variables locales de un método, las cláusulas de manejo de excepciones, el tamaño de la pila y el IL en bruto. ¡Es algo así como la inversa de Reflection.Emit!

Inspeccionar el IL en bruto de un método puede ser útil para perfilar el código. Un uso sencillo sería determinar qué métodos en un ensamblado han cambiado cuando se actualiza dicho ensamblado.

Para ilustrar el análisis de IL, escribiremos una aplicación que desensambla IL al estilo de ildasm. Esto podría ser utilizado como punto de partida para una herramienta de análisis de código o un desensamblador de un lenguaje de nivel superior.



Recuerda que en la API de reflexión, todos los constructos funcionales de C# se representan mediante un subtipo de MethodBase o (en el caso de propiedades, eventos e indexadores) tienen objetos MethodBase adjuntos a ellos.

Escribiendo un desensamblador

Aquí tienes un ejemplo de la salida que nuestro desensamblador producirá:

```
IL_00EB:  ldfld       Disassembler._pos
IL_00F0:  ldloc.2
IL_00F1:  add
IL_00F2:  ldelema     System.Byte
IL_00F7:  ldstr       "Hello world"
IL_00FC:  call        System.Byte.ToString
IL_0101:  ldstr       " "
IL_0106:  call        System.String.Concat
```

Para obtener esta salida, debemos analizar los tokens binarios que componen el IL. El primer paso es llamar al método GetILAsByteArray en MethodBody para obtener el IL como una matriz de bytes. Para facilitar el resto del trabajo, lo escribiremos en una clase de la siguiente manera:

```
public class Disassembler
{
  public static string Disassemble (MethodBase method)
    => new Disassembler (method).Dis();

  StringBuilder _output;    // The result to which we'll keep appending
  Module _module;           // This will come in handy later
  byte[] _il;               // The raw byte code
  int _pos;                 // The position we're up to in the byte code

  Disassembler (MethodBase method)
  {
    _module = method.DeclaringType.Module;
    _il = method.GetMethodBody().GetILAsByteArray();
  }

  string Dis()
  {
    _output = new StringBuilder();
    while (_pos < _il.Length) DisassembleNextInstruction();
    return _output.ToString();
  }
}
```

El método estático "Disassemble" será el único miembro público de esta clase. Todos los demás miembros serán privados y formarán parte del proceso de desensamblado. El método "Dis" contiene el bucle principal donde procesamos cada instrucción.

Con esta estructura en su lugar, lo único que queda por hacer es escribir "DisassembleNextInstruction". Pero antes de hacerlo, será útil cargar todos los códigos de operación en un diccionario estático para poder acceder a ellos por su valor de 8 o 16 bits. La forma más sencilla de lograr esto es utilizar la reflexión para obtener todos los campos estáticos cuyo tipo sea "OpCode" en la clase "OpCodes":

```
static Dictionary<short,OpCode> _opcodes = new Dictionary<short,OpCode>();

static Disassembler()
{
  Dictionary<short, OpCode> opcodes = new Dictionary<short, OpCode>();
    foreach (FieldInfo fi in typeof (OpCodes).GetFields
                             (BindingFlags.Public | BindingFlags.Static))
      if (typeof (OpCode).IsAssignableFrom (fi.FieldType))
      {
        OpCode code = (OpCode) fi.GetValue (null);   // Get field's value
        if (code.OpCodeType != OpCodeType.Nternal)
          _opcodes.Add (code.Value, code);
      }
}
```

Lo hemos escrito en un constructor estático para que se ejecute solo una vez.

Ahora podemos escribir "DisassembleNextInstruction". Cada instrucción de IL consta de un código de operación de uno o dos bytes, seguido de un operando de cero, uno, dos, cuatro o ocho bytes (a excepción de los códigos de operación "inline switch", que están seguidos por un número variable de operandos). Por lo tanto, leemos el código de operación y luego el operando, y luego escribimos el resultado:

```
void DisassembleNextInstruction()
{
  int opStart = _pos;

  OpCode code = ReadOpCode();
  string operand = ReadOperand (code);

   _output.AppendFormat ("IL_{0:X4}:  {1,-12} {2}",
                         opStart, code.Name, operand);
   _output.AppendLine();
 }
```

Para leer un opcode, avanzamos un byte y verificamos si tenemos una instrucción válida. Si no es así, avanzamos otro byte y buscamos una instrucción de dos bytes:

```
OpCode ReadOpCode()
{
  byte byteCode = _il [_pos++];
  if (_opcodes.ContainsKey (byteCode)) return _opcodes [byteCode];

  if (_pos == _il.Length)  throw new Exception ("Unexpected end of IL");

  short shortCode = (short) (byteCode * 256 + _il [_pos++]);

  if (!_opcodes.ContainsKey (shortCode))
    throw new Exception ("Cannot find opcode " + shortCode);

  return _opcodes [shortCode];
}
```

Para leer un operando, primero debemos establecer su longitud. Podemos hacer esto en función del tipo de operando. Dado que la mayoría de los operandos tienen una longitud de cuatro bytes, podemos filtrar las excepciones de manera bastante sencilla en una cláusula condicional.

El siguiente paso es llamar a FormatOperand, que intenta formatear el operando:

```
string ReadOperand (OpCode c)
{
  int operandLength =
    c.OperandType == OperandType.InlineNone
      ? 0 :
    c.OperandType == OperandType.ShortInlineBrTarget ||
    c.OperandType == OperandType.ShortInlineI ||
    c.OperandType == OperandType.ShortInlineVar
      ? 1 :
    c.OperandType == OperandType.InlineVar
      ? 2 :
    c.OperandType == OperandType.InlineI8 ||
    c.OperandType == OperandType.InlineR
      ? 8 :
    c.OperandType == OperandType.InlineSwitch
      ? 4 * (BitConverter.ToInt32 (_il, _pos) + 1) :
      4;  // All others are 4 bytes

  if (_pos + operandLength > _il.Length)
    throw new Exception ("Unexpected end of IL");

  string result = FormatOperand (c, operandLength);
  if (result == null)
  {                        // Write out operand bytes in hex
    result = "";
    for (int i = 0; i < operandLength; i++)
      result += _il [_pos + i].ToString ("X2") + " ";

  }
  _pos += operandLength;
  return result;
}
```

Si el resultado de llamar a FormatOperand es nulo, significa que el operando no necesita ningún formato especial, por lo que simplemente lo escribimos en hexadecimal. Podríamos probar el desensamblador en este punto escribiendo un método FormatOperand que siempre devuelva nulo. Aquí está cómo se vería la salida:

```
IL_00A8:  ldfld       98 00 00 04
IL_00AD:  ldloc.2
IL_00AE:  add
IL_00AF:  ldelema     64 00 00 01
IL_00B4:  ldstr       26 04 00 70
IL_00B9:  call        B6 00 00 0A
IL_00BE:  ldstr       11 01 00 70
IL_00C3:  call        91 00 00 0A
...
```

Aunque los opcodes son correctos, los operandos no son muy útiles. En lugar de números hexadecimales, queremos nombres de miembros y cadenas de texto. El método

FormatOperand, cuando se escriba, abordará esto, identificando los casos especiales que se benefician de dicho formato. Estos casos comprenden la mayoría de los operandos de cuatro bytes y las instrucciones de salto corto.

```
string FormatOperand (OpCode c, int operandLength)
{
  if (operandLength == 0) return "";

  if (operandLength == 4)
    return Get4ByteOperand (c);
  else if (c.OperandType == OperandType.ShortInlineBrTarget)
    return GetShortRelativeTarget();
  else if (c.OperandType == OperandType.InlineSwitch)
    return GetSwitchTarget (operandLength);
  else
    return null;
}
```

Hay tres tipos de operandos de cuatro bytes que tratamos de manera especial. El primero son las referencias a miembros o tipos, con estos extraemos el nombre del miembro o tipo llamando al método ResolveMember del módulo definitorio. El segundo caso son las cadenas de texto, que se almacenan en los metadatos del módulo del ensamblado y se pueden recuperar llamando a ResolveString. El último caso son los objetivos de los saltos, donde el operando se refiere a un desplazamiento en bytes en el IL. Formateamos estos trabajando la dirección absoluta después de la instrucción actual (+ cuatro bytes):

```
string Get4ByteOperand (OpCode c)
{
  int intOp = BitConverter.ToInt32 (_il, _pos);

  switch (c.OperandType)
  {

     case OperandType.InlineTok:
     case OperandType.InlineMethod:
     case OperandType.InlineField:
     case OperandType.InlineType:
       MemberInfo mi;
       try   { mi = _module.ResolveMember (intOp); }
       catch { return null; }
       if (mi == null) return null;

       if (mi.ReflectedType != null)
         return mi.ReflectedType.FullName + "." + mi.Name;
       else if (mi is Type)
         return ((Type)mi).FullName;
       else
         return mi.Name;

     case OperandType.InlineString:
       string s = _module.ResolveString (intOp);
       if (s != null) s = "'" + s + "'";
       return s;

     case OperandType.InlineBrTarget:
       return "IL_" + (_pos + intOp + 4).ToString ("X4");

     default:
       return null;
   }
 }
```



El punto en el que llamamos a ResolveMember es una buena oportunidad para una herramienta de análisis de código que informa sobre las dependencias de los métodos.

Para cualquier otro opcode de cuatro bytes, devolvemos nulo (esto hará que ReadOperand formatee el operando como dígitos hexadecimales).

Los últimos tipos de operandos que requieren atención especial son los destinos de salto corto y los switches en línea. Un destino de salto corto describe la posición de destino como un solo byte con signo, al final de la instrucción actual (es decir, + un byte). Un switch target va seguido de un número variable de destinos de salto de cuatro bytes:

```
string GetShortRelativeTarget()
{
  int absoluteTarget = _pos + (sbyte) _il [_pos] + 1;
  return "IL_" + absoluteTarget.ToString ("X4");
}

string GetSwitchTarget (int operandLength)
{
  int targetCount = BitConverter.ToInt32 (_il, _pos);
  string [] targets = new string [targetCount];
  for (int i = 0; i < targetCount; i++)
  {
    int ilTarget = BitConverter.ToInt32 (_il, _pos + (i + 1) * 4);
    targets [i] = "IL_" + (_pos + ilTarget + operandLength).ToString ("X4");

  }
  return "(" + string.Join (", ", targets) + ")";
}
```

Esto completa el desensamblador. Podemos probarlo desensamblando uno de sus propios métodos:

```
MethodInfo mi = typeof (Disassembler).GetMethod (
  "ReadOperand", BindingFlags.Instance | BindingFlags.NonPublic);

Console.WriteLine (Disassembler.Disassemble (mi));
```

19 Programación dinámica (Dynamic Programming)

El Capítulo 4 explicó cómo funciona el enlace dinámico en el lenguaje C#. En este capítulo, examinamos brevemente el tiempo de ejecución de lenguajes dinámicos (DLR, por sus siglas en inglés) y luego exploramos los siguientes patrones de programación dinámica:

- Unificación de tipos numéricos
- Resolución de sobrecarga de miembros dinámicos
- Enlace personalizado (implementación de objetos dinámicos)
- Interoperabilidad entre lenguajes dinámicos



En el Capítulo 24, describimos cómo el uso de "dynamic" puede mejorar la interoperabilidad con COM.

Los tipos en este capítulo residen en el espacio de nombres System.Dynamic, excepto CallSite<>, que reside en System.Runtime.CompilerServices.

El tiempo de ejecución de lenguajes dinámicos (The Dynamic Language Runtime)

C# se basa en el DLR para realizar el enlace dinámico.

A diferencia de su nombre, el DLR no es una versión dinámica del CLR. En cambio, es una biblioteca que se encuentra sobre el CLR, al igual que cualquier otra biblioteca como System.Xml.dll. Su función principal es proporcionar servicios de tiempo de ejecución para unificar la programación dinámica, tanto en lenguajes de tipos estáticos como dinámicos. Por lo tanto, lenguajes como C#, Visual Basic, IronPython e IronRuby utilizan el mismo protocolo para llamar funciones de manera dinámica. Esto les permite compartir bibliotecas y llamar código escrito en otros lenguajes.

El DLR también facilita la escritura de nuevos lenguajes dinámicos en .NET. En lugar de tener que emitir Lenguaje Intermedio (IL), los autores de lenguajes dinámicos trabajan a nivel de árboles de expresión (los mismos árboles de expresión en System.Linq.Expressions de los que hablamos en el Capítulo 8).

Además, el DLR asegura que todos los consumidores se beneficien del almacenamiento en caché en el sitio de llamada, una optimización mediante la cual el DLR evita repetir innecesariamente las decisiones de resolución de miembros potencialmente costosas realizadas durante el enlace dinámico.

Unificación de tipos numéricos

El Capítulo 4 explicó cómo "dynamic" nos permite escribir un único método que funciona con todos los tipos numéricos:

```
static dynamic Mean (dynamic x, dynamic y) => (x + y) / 2;

static void Main()
{
  int x = 3, y = 5;
  Console.WriteLine (Mean (x, y));
}
```

¿Qué son los sitios de llamada?

Cuando el compilador encuentra una expresión dinámica, no tiene idea de quién evaluará esa expresión en tiempo de ejecución. Por ejemplo, considera el siguiente método:

```
public dynamic Foo (dynamic x, dynamic y)
{
  return x / y;   // Dynamic expression
}
```

Las variables x e y podrían ser cualquier objeto CLR, un objeto COM o incluso un objeto alojado en un lenguaje dinámico. El compilador no puede seguir su enfoque estático habitual de emitir una llamada a un método conocido de un tipo conocido. En su lugar, el compilador emite código que eventualmente resulta en un árbol de expresiones que describe la operación, gestionado por un sitio de llamada que el DLR enlazará en tiempo de ejecución. El sitio de llamada actúa como intermediario entre el llamante y el llamado.

Un sitio de llamada se representa mediante la clase CallSite<> en System.Core.dll. Podemos ver esto desensamblando el método anterior, y el resultado sería algo como esto:

```
static CallSite<Func<CallSite,object,object,object>> divideSite;

[return: Dynamic]
public object Foo ([Dynamic] object x, [Dynamic] object y)
{
  if (divideSite == null)
    divideSite =
      CallSite<Func<CallSite,object,object,object>>.Create (
        Microsoft.CSharp.RuntimeBinder.Binder.BinaryOperation (
          CSharpBinderFlags.None,
          ExpressionType.Divide,
          /* Remaining arguments omitted for brevity */ ));

  return divideSite.Target (divideSite, x, y);
}
```

Como puedes ver, el sitio de llamada se almacena en caché en un campo estático para evitar el costo de volver a crearlo en cada llamada. El DLR también almacena en caché el resultado de la fase de enlace y los objetivos reales del método. (Puede haber múltiples objetivos dependiendo de los tipos de x e y).

La llamada dinámica real ocurre llamando al Target del sitio (un delegado), pasando los operandos x e y.

Observa que la clase Binder es específica de C#. Cada lenguaje con soporte para enlace dinámico proporciona un enlazador específico del lenguaje para ayudar al DLR a interpretar las expresiones de manera específica para ese lenguaje, para no sorprender al programador. Por

ejemplo, si llamamos a Foo con valores enteros de 5 y 2, el enlazador de C# se aseguraría de que obtuviéramos 2. En cambio, un enlazador de VB.NET nos daría 2.5.



Es una reflexión humorística sobre C# que las palabras clave "static" y "dynamic" puedan aparecer juntas. Lo mismo se aplica a las palabras clave "internal" y "extern".

Sin embargo, esto sacrifica (innecesariamente) la seguridad de tipo estática. Lo siguiente se compila sin errores pero luego falla en tiempo de ejecución:

```
string s = Mean (3, 5);   // Runtime error!
```

Podemos solucionar esto introduciendo un parámetro genérico y luego haciendo una conversión a dynamic dentro del cálculo mismo:

```
static T Mean<T> (T x, T y)
{
  dynamic result = ((dynamic) x + y) / 2;
  return (T) result;
}
```

Observa que hacemos una conversión explícita del resultado a T. Si omitiéramos esta conversión, estaríamos confiando en una conversión implícita, que podría parecer funcionar correctamente al principio. Sin embargo, la conversión implícita fallaría en tiempo de ejecución al llamar al método con un tipo integral de 8 o 16 bits. Para entender por qué, considera lo que sucede con la tipificación estática ordinaria cuando sumas dos números de 8 bits juntos:

```
byte b = 3;
Console.WriteLine ((b + b).GetType().Name);  // Int32
```

Obtenemos un Int32 porque el compilador "promociona" los números de 8 o 16 bits a Int32 antes de realizar operaciones aritméticas. Por coherencia, el enlazador de C# instruye al DLR para que haga exactamente lo mismo, y obtenemos un Int32 que requiere una conversión explícita al tipo numérico más pequeño. Por supuesto, esto podría crear la posibilidad de desbordamiento si, por ejemplo, estuviéramos sumando en lugar de promediar los valores.

El enlace dinámico incurre en una pequeña pérdida de rendimiento, incluso con la caché del sitio de llamada. Puedes mitigar esto agregando sobrecargas con tipos estáticamente tipados que cubran solo los tipos más comúnmente utilizados. Por ejemplo, si el perfilado de rendimiento posterior mostrara que llamar a Mean con valores double era un cuello de botella, podrías agregar la siguiente sobrecarga:

```
static double Mean (double x, double y) => (x + y) / 2;
```

El compilador favorecerá esa sobrecarga cuando se llame a Mean con argumentos que se conocen en tiempo de compilación como de tipo double.

Resolución de sobrecarga de miembros dinámicos

Llamar a un método estáticamente conocido con argumentos de tipo dinámico pospone la resolución de sobrecarga de miembros desde el tiempo de compilación hasta el tiempo de ejecución. Esto es útil para simplificar ciertas tareas de programación, como simplificar el patrón de diseño Visitor. También es útil para superar las limitaciones impuestas por la tipificación estática de C#.

Simplificación del patrón Visitor

En esencia, el patrón Visitor permite "añadir" un método a una jerarquía de clases sin alterar las clases existentes. Aunque útil, este patrón en su encarnación estática es sutil e intuitivo en comparación con la mayoría de los otros patrones de diseño. También requiere que las clases visitadas sean "amigables para el visitante" al exponer un método Accept, lo cual puede ser imposible si las clases no están bajo tu control.

Con la vinculación dinámica, puedes lograr el mismo objetivo de manera más sencilla y sin necesidad de modificar las clases existentes. Para ilustrarlo, considera la siguiente jerarquía de clases:

```
class Person
{
  public string FirstName { get; set; }
  public string LastName  { get; set; }

  // The Friends collection may contain Customers & Employees:
  public readonly IList<Person> Friends = new Collection<Person> ();
}

class Customer : Person { public decimal CreditLimit { get; set; } }
class Employee : Person { public decimal Salary      { get; set; } }
```

Supongamos que queremos escribir un método que exporte de forma programática los detalles de una persona a un XElement XML. La solución más obvia sería escribir un método virtual llamado ToXElement() en la clase Person que devuelva un XElement poblado con las propiedades de la persona. Luego lo sobrescribiríamos en las clases Customer y Employee para que también se poblara el XElement con CreditLimit y Salary. Sin embargo, este patrón puede ser problemático por dos razones:

- Es posible que no seamos propietarios de las clases Person, Customer y Employee, lo que haría imposible agregar métodos a ellas. (Y los métodos de extensión no brindarían comportamiento polimórfico).
- Las clases Person, Customer y Employee podrían ser bastante grandes. Un antipatrón frecuente es el "Objeto Dios" (God Object), en el cual una clase como Person atrae tanta funcionalidad que se convierte en una pesadilla de mantener. Un buen antídoto es evitar agregar funciones a la clase Person que no necesitan acceder al estado privado de Person. Un método ToXElement podría ser un excelente candidato.

Con la resolución dinámica de sobrecarga de miembros, podemos escribir la funcionalidad de ToXElement en una clase separada, sin recurrir a interruptores feos basados en el tipo:

```
class ToXElementPersonVisitor
{
  public XElement DynamicVisit (Person p) => Visit ((dynamic)p);

  XElement Visit (Person p)
  {
    return new XElement ("Person",
      new XAttribute ("Type", p.GetType().Name),
      new XElement ("FirstName", p.FirstName),
      new XElement ("LastName", p.LastName),
      p.Friends.Select (f => DynamicVisit (f))
    );
  }

  XElement Visit (Customer c)   // Specialized logic for customers
  {
    XElement xe = Visit ((Person)c);   // Call "base" method
    xe.Add (new XElement ("CreditLimit", c.CreditLimit));
    return xe;
  }

  XElement Visit (Employee e)   // Specialized logic for employees
  {
    XElement xe = Visit ((Person)e);   // Call "base" method
    xe.Add (new XElement ("Salary", e.Salary));
    return xe;
  }
}
```

El método DynamicVisit realiza una llamada de despacho dinámico, llamando a la versión más específica de Visit según se determine en tiempo de ejecución. Observa la línea en negrita, en la que llamamos a DynamicVisit en cada persona de la colección Friends. Esto asegura que si un amigo es un Cliente o un Empleado, se llame a la sobrecarga correcta.

Podemos demostrar esta clase de la siguiente manera:

```
var cust = new Customer
{
  FirstName = "Joe", LastName = "Bloggs", CreditLimit = 123
};
cust.Friends.Add (
  new Employee { FirstName = "Sue", LastName = "Brown", Salary = 50000 }
);

Console.WriteLine (new ToXElementPersonVisitor().DynamicVisit (cust));
```

Aquí está el resultado:

```
<Person Type="Customer">
  <FirstName>Joe</FirstName>
  <LastName>Bloggs</LastName>
  <Person Type="Employee">
    <FirstName>Sue</FirstName>
    <LastName>Brown</LastName>
    <Salary>50000</Salary>
  </Person>
  <CreditLimit>123</CreditLimit>
</Person>
```

Variaciones

Si planeas más de una clase de visitante, una variación útil es definir una clase base abstracta para los visitantes:

```
abstract class PersonVisitor<T>
{
  public T DynamicVisit (Person p) { return Visit ((dynamic)p); }

  protected abstract T Visit (Person p);
  protected virtual T Visit (Customer c) { return Visit ((Person) c); }
  protected virtual T Visit (Employee e) { return Visit ((Person) e); }
}
```

Las subclases no necesitan definir su propio método DynamicVisit: todo lo que hacen es anular las versiones de Visit cuyo comportamiento desean especializar. Esto también tiene las ventajas de centralizar los métodos que abarcan la jerarquía de Person y permitir que los implementadores llamen a los métodos base de manera más natural:

```
class ToXElementPersonVisitor : PersonVisitor<XElement>
{
  protected override XElement Visit (Person p)
  {
    return new XElement ("Person",
      new XAttribute ("Type", p.GetType().Name),
      new XElement ("FirstName", p.FirstName),
      new XElement ("LastName", p.LastName),
      p.Friends.Select (f => DynamicVisit (f))
    );
  }

  protected override XElement Visit (Customer c)
  {
    XElement xe = base.Visit (c);
    xe.Add (new XElement ("CreditLimit", c.CreditLimit));
    return xe;
  }

  protected override XElement Visit (Employee e)
  {
    XElement xe = base.Visit (e);
    xe.Add (new XElement ("Salary", e.Salary));
    return xe;

    }
  }
```

Incluso puedes crear una subclase de ToXElementPersonVisitor.

Llamando anónimamente a miembros de un tipo genérico

La rigidez del tipado estático de C# es una espada de doble filo. Por un lado, impone un grado de corrección en tiempo de compilación. Por otro lado, ocasionalmente dificulta o incluso hace

imposible expresar ciertos tipos de código, en cuyo caso debes recurrir a la reflexión. En estas situaciones, el enlace dinámico es una alternativa más limpia y rápida que la reflexión.

Un ejemplo es cuando necesitas trabajar con un objeto de tipo G<T> donde T es desconocido. Podemos ilustrar esto definiendo la siguiente clase:

```
public class Foo<T> { public T Value; }
```

Supongamos que luego escribimos un método de la siguiente manera:

```
static void Write (object obj)
{
  if (obj is Foo<>)                            // Illegal
    Console.WriteLine ((Foo<>) obj).Value);    // Illegal
}
```

Este método no se compilará: no puedes invocar miembros de tipos genéricos sin enlazarlos.

El enlace dinámico ofrece dos formas de solucionar esto. La primera es acceder al miembro Value de forma dinámica de la siguiente manera:

```
static void Write (dynamic obj)
{
  try { Console.WriteLine (obj.Value); }
  catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException) {...}
}
```

Despacho Múltiple (Multiple Dispatch)

C# y el CLR siempre han admitido una forma limitada de dinamismo en forma de llamadas a métodos virtuales. Esto difiere del enlace dinámico de C# en que, para las llamadas a métodos virtuales, el compilador debe comprometerse con un miembro virtual en particular en tiempo de compilación, basándose en el nombre y la firma del miembro que llamaste. Esto significa que:

• La expresión de llamada debe ser completamente entendida por el compilador (por ejemplo, debe decidir en tiempo de compilación si un miembro objetivo es un campo o una propiedad).

• La resolución de sobrecarga debe completarse completamente por el compilador, basándose en los tipos de argumentos en tiempo de compilación.

Una consecuencia de este último punto es que la capacidad de realizar llamadas a métodos virtuales se conoce como despacho único. Para entender por qué, considera la siguiente llamada de método (en la que Walk es un método virtual):

```
animal.Walk (owner);
```

La decisión en tiempo de ejecución de invocar el método Walk de un perro o el método Walk de un gato depende solo del tipo del receptor, animal (por eso se llama "único"). Si muchas sobrecargas de Walk aceptan diferentes tipos de owner, se seleccionará una sobrecarga en tiempo de compilación sin tener en cuenta el tipo de tiempo de ejecución del objeto owner. En otras palabras, solo el tipo de tiempo de ejecución del receptor puede variar qué método se llama.

En contraste, una llamada dinámica pospone la resolución de sobrecarga hasta tiempo de ejecución:

```
animal.Walk ((dynamic) owner);
```

La elección final de qué método Walk llamar ahora depende de los tipos tanto de animal como de owner. Esto se llama despacho múltiple porque los tipos de tiempo de ejecución de los argumentos, además del tipo de receptor, contribuyen a la determinación de qué método Walk llamar.

Esto tiene la ventaja (potencial) de funcionar con cualquier objeto que defina un campo o propiedad llamada "Value". Sin embargo, hay un par de problemas. Primero, capturar una excepción de esta manera es un poco complicado e ineficiente (y no hay forma de preguntarle al DLR de antemano, "¿Tendrá éxito esta operación?"). Segundo, este enfoque no funcionaría si Foo fuera una interfaz (digamos, IFoo<T>) y cualquiera de las siguientes condiciones fueran verdaderas:

- "Value" estuviera implementado de forma explícita.
- El tipo que implementó IFoo<T> fuera inaccesible (más sobre esto en breve).

Una mejor solución es escribir un método de ayuda sobrecargado llamado GetFooValue y llamarlo utilizando la resolución de sobrecarga de miembros dinámicos:

```
static void Write (dynamic obj)
{
  object result = GetFooValue (obj);
  if (result != null) Console.WriteLine (result);
}

static T GetFooValue<T> (Foo<T> foo) => foo.Value;
static object GetFooValue (object foo) => null;
```

Observa que hemos sobrecargado GetFooValue para aceptar un parámetro de tipo object, que actúa como una alternativa para cualquier tipo. En tiempo de ejecución, el enlazador dinámico de C# elegirá la mejor sobrecarga al llamar a GetFooValue con un argumento dinámico. Si el objeto en cuestión no se basa en Foo<T>, elegirá la sobrecarga del parámetro object en lugar de lanzar una excepción.



Una alternativa es escribir solo la primera sobrecarga de GetFooValue y luego capturar la excepción RuntimeBinderException. La ventaja es que se distingue el caso de "foo.Value" siendo nulo. La desventaja es que incurre en la sobrecarga de rendimiento de lanzar y capturar una excepción.

En el Capítulo 18, resolvimos el mismo problema con una interfaz utilizando reflexión, con mucho más esfuerzo (ver "Llamando anónimamente a miembros de una interfaz genérica" en la página 800). El ejemplo que utilizamos fue diseñar una versión más poderosa de ToString() que pudiera comprender objetos como IEnumerable e IGrouping<,>.

Aquí está el mismo ejemplo resuelto de manera más elegante utilizando el enlace dinámico:

```
static string GetGroupKey<TKey,TElement> (IGrouping<TKey,TElement> group)
  => "Group with key=" + group.Key + ": ";

static string GetGroupKey (object source) => null;

public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  if (value is string s) return s;
  if (value.GetType().IsPrimitive) return value.ToString();

  StringBuilder sb = new StringBuilder();

  string groupKey = GetGroupKey ((dynamic)value);   // Dynamic dispatch
  if (groupKey != null) sb.Append (groupKey);

  if (value is IEnumerable)
    foreach (object element in ((IEnumerable)value))
      sb.Append (ToStringEx (element) + " ");

  if (sb.Length == 0) sb.Append (value.ToString());

  return "\r\n" + sb.ToString();
}
```

Aquí está en acción:

```
Console.WriteLine (ToStringEx ("xyyzzz".GroupBy (c => c) ));

Group with key=x: x
Group with key=y: y y
Group with key=z: z z z
```

Observa que utilizamos la resolución de sobrecarga de miembros dinámicos para resolver este problema. Si en su lugar hiciéramos lo siguiente:

```
dynamic d = value;
try { groupKey = d.Value); }
catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException) {...}
```

fallaría porque el operador GroupBy de LINQ devuelve un tipo que implementa IGrouping<,>, que a su vez es interno y, por lo tanto, inaccesible:

```
internal class Grouping : IGrouping<TKey,TElement>, ...
{
  public TKey Key;
  ...
}
```

A pesar de que la propiedad Key se declara como pública, su clase contenedora la limita a interna, lo que la hace accesible solo a través de la interfaz IGrouping<,>. Y como se explica en el Capítulo 4, no hay forma de instruir al DLR para que se enlace con esa interfaz al invocar dinámicamente el miembro Value.

Implementación de objetos dinámicos

Un objeto puede proporcionar su semántica de enlace implementando IDynamicMetaObjectProvider, o de manera más sencilla, mediante la subclase DynamicObject, que proporciona una implementación predeterminada de esta interfaz. Esto se demuestra brevemente en el Capítulo 4 a través del siguiente ejemplo:

```
dynamic d = new Duck();
d.Quack();                  // Quack method was called
d.Waddle();                 // Waddle method was called

public class Duck : DynamicObject
{
  public override bool TryInvokeMember (
    InvokeMemberBinder binder, object[] args, out object result)
  {
    Console.WriteLine (binder.Name + " method was called");
    result = null;
    return true;
  }
}
```

DynamicObject

En el ejemplo anterior, sobrescribimos TryInvokeMember, que permite al consumidor invocar un método en el objeto dinámico, como Quack o Waddle. DynamicObject expone otros métodos virtuales que permiten a los consumidores utilizar otros constructos de programación también. Los siguientes corresponden a constructos que tienen representaciones en C#:

Method	Programming construct
`TryInvokeMember`	Method
`TryGetMember, TrySetMember`	Property or field
`TryGetIndex, TrySetIndex`	Indexer
`TryUnaryOperation`	Unary operator such as !
`TryBinaryOperation`	Binary operator such as ==
`TryConvert`	Conversion (cast) to another type
`TryInvoke`	Invocation on the object itself—e.g., `d("foo")`

Estos métodos deben devolver true si tienen éxito. Si devuelven false, el DLR volverá al enlazador de lenguaje y buscará un miembro coincidente en el objeto dinámico (subclase) en sí mismo. Si esto falla, se lanzará una excepción RuntimeBinderException.

Podemos ilustrar TryGetMember y TrySetMember con una clase que nos permite acceder dinámicamente a un atributo en un XElement (System.Xml.Linq):

```
static class XExtensions
{
  public static dynamic DynamicAttributes (this XElement e)
    => new XWrapper (e);

  class XWrapper : DynamicObject
  {
    XElement _element;
    public XWrapper (XElement e) { _element = e; }

    public override bool TryGetMember (GetMemberBinder binder,
                                       out object result)
    {
      result = _element.Attribute (binder.Name).Value;
      return true;
    }

    public override bool TrySetMember (SetMemberBinder binder,
                                       object value)
    {
      _element.SetAttributeValue (binder.Name, value);
      return true;
    }
  }
}
```

Así es cómo se usa:

```
XElement x = XElement.Parse (@"<Label Text=""Hello"" Id=""5""/>");
dynamic da = x.DynamicAttributes();
Console.WriteLine (da.Id);           // 5
da.Text = "Foo";
Console.WriteLine (x.ToString());    // <Label Text="Foo" Id="5" />
```

Lo siguiente hace algo similar para System.Data.IDataRecord, lo que facilita el uso de los lectores de datos (data readers):

```
public class DynamicReader : DynamicObject
{
  readonly IDataRecord _dataRecord;
  public DynamicReader (IDataRecord dr) { _dataRecord = dr; }

  public override bool TryGetMember (GetMemberBinder binder,
                                     out object result)
  {
    result = _dataRecord [binder.Name];
    return true;
  }
```

```
}
...
using (IDataReader reader = someDbCommand.ExecuteReader())
{
  dynamic dr = new DynamicReader (reader);
  while (reader.Read())
  {
    int id = dr.ID;
    string firstName = dr.FirstName;
    DateTime dob = dr.DateOfBirth;
    ...
  }
}
```

La siguiente demostración muestra el uso de TryBinaryOperation y TryInvoke:

```
dynamic d = new Duck();
Console.WriteLine (d + d);          // foo
Console.WriteLine (d (78, 'x'));    // 123

public class Duck : DynamicObject
{
  public override bool TryBinaryOperation (BinaryOperationBinder binder,
                                           object arg, out object result)
  {
    Console.WriteLine (binder.Operation);   // Add
    result = "foo";
    return true;
  }

  public override bool TryInvoke (InvokeBinder binder,
                                  object[] args, out object result)
  {
    Console.WriteLine (args[0]);    // 78
    result = 123;
    return true;
  }
}
```

DynamicObject también expone algunos métodos virtuales para el beneficio de los lenguajes dinámicos. En particular, sobrescribir GetDynamicMemberNames te permite devolver una lista de todos los nombres de miembros que tu objeto dinámico proporciona.



Otra razón para implementar GetDynamicMemberNames es que el depurador de Visual Studio lo utiliza para mostrar una vista de un objeto dinámico.

ExpandoObject

Otra aplicación sencilla de DynamicObject sería escribir una clase dinámica que almacene y recupere objetos en un diccionario, utilizando cadenas como clave. Sin embargo, esta funcionalidad ya está proporcionada a través de la clase ExpandoObject:

```
dynamic x = new ExpandoObject();
x.FavoriteColor = ConsoleColor.Green;
x.FavoriteNumber = 7;
Console.WriteLine (x.FavoriteColor);     // Green
Console.WriteLine (x.FavoriteNumber);    // 7
```

ExpandoObject implementa IDictionary<string,object>, por lo que podemos continuar nuestro ejemplo y hacer lo siguiente:

```
var dict = (IDictionary<string,object>) x;
Console.WriteLine (dict ["FavoriteColor"]);     // Green
Console.WriteLine (dict ["FavoriteNumber"]);    // 7
Console.WriteLine (dict.Count);                 // 2
```

Interoperabilidad con lenguajes dinámicos

Aunque C# admite la vinculación dinámica a través de la palabra clave dynamic, no permite ejecutar una expresión descrita en una cadena en tiempo de ejecución:

```
string expr = "2 * 3";
// We can't "execute" expr
```

Esto se debe a que el código para traducir una cadena en un árbol de expresiones requiere un analizador léxico y semántico. Estas características están incorporadas en el compilador de C# y no están disponibles como un servicio en tiempo de ejecución. En tiempo de ejecución, C# simplemente proporciona un enlazador (binder) que instruye al DLR sobre cómo interpretar un árbol de expresiones ya construido.

Los verdaderos lenguajes dinámicos como IronPython e IronRuby sí permiten ejecutar una cadena arbitraria, y esto es útil en tareas como scripting, configuración dinámica e implementación de motores de reglas dinámicas. Por lo tanto, aunque puedes escribir la mayor parte de tu aplicación en C#, puede ser útil llamar a un lenguaje dinámico para este tipo de tareas. Además, es posible que desees utilizar una API escrita en un lenguaje dinámico donde no haya una funcionalidad equivalente disponible en una biblioteca de .NET.



El paquete NuGet de Roslyn scripting Microsoft.CodeAnalysis.CSharp.Scripting proporciona una API que te permite ejecutar una cadena de C#, aunque lo hace compilando tu código en un programa. La sobrecarga de compilación hace que sea más lento que la interoperabilidad con Python, a menos que tengas la intención de ejecutar la misma expresión repetidamente.

En el siguiente ejemplo, usamos IronPython para evaluar una expresión creada en tiempo de ejecución desde C#. Podrías usar el siguiente script para escribir una calculadora.



Para ejecutar este código, agrega los paquetes NuGet DynamicLanguageRuntime (no confundir con el paquete System.Dynamic.Runtime) e IronPython a tu aplicación.

```
using System;
using IronPython.Hosting;
using Microsoft.Scripting;
using Microsoft.Scripting.Hosting;

int result = (int) Calculate ("2 * 3");
Console.WriteLine (result);              // 6

object Calculate (string expression)
{
  ScriptEngine engine = Python.CreateEngine();
  return engine.Execute (expression);
}
```

Debido a que estamos pasando una cadena a Python, la expresión se evaluará según las reglas de Python y no las de C#. Esto también significa que podemos utilizar las características del lenguaje de Python, como las listas:

```
var list = (IEnumerable) Calculate ("[1, 2, 3] + [4, 5]");
foreach (int n in list) Console.Write (n);  // 12345
```

Para pasar variables desde C# a Python, se requieren algunos pasos adicionales. El siguiente ejemplo ilustra esos pasos y podría ser la base de un motor de reglas:

```
// The following string could come from a file or database:
string auditRule = "taxPaidLastYear / taxPaidThisYear > 2";

ScriptEngine engine = Python.CreateEngine ();

ScriptScope scope = engine.CreateScope ();
scope.SetVariable ("taxPaidLastYear", 20000m);
scope.SetVariable ("taxPaidThisYear", 8000m);

ScriptSource source = engine.CreateScriptSourceFromString (
                      auditRule, SourceCodeKind.Expression);

bool auditRequired = (bool) source.Execute (scope);
Console.WriteLine (auditRequired);   // True
```

También puedes obtener variables llamando a `GetVariable`:

```
string code = "result = input * 3";

ScriptEngine engine = Python.CreateEngine();

ScriptScope scope = engine.CreateScope();
scope.SetVariable ("input", 2);

ScriptSource source = engine.CreateScriptSourceFromString (code,
                                  SourceCodeKind.SingleStatement);
source.Execute (scope);
Console.WriteLine (scope.GetVariable ("result"));   // 6
```

Observa que en el segundo ejemplo especificamos `SourceCodeKind.SingleStatement` en lugar de `Expression` para informar al motor que queremos ejecutar una instrucción.

Los tipos se transmiten automáticamente entre el mundo .NET y Python. Incluso puedes acceder a los miembros de objetos .NET desde el lado de scripting:

```
string code = @"sb.Append (""World"")";

ScriptEngine engine = Python.CreateEngine ();

ScriptScope scope = engine.CreateScope ();
var sb = new StringBuilder ("Hello");
scope.SetVariable ("sb", sb);

ScriptSource source = engine.CreateScriptSourceFromString (
                      code, SourceCodeKind.SingleStatement);
source.Execute (scope);
Console.WriteLine (sb.ToString());   // HelloWorld
```

20 Criptografía

En este capítulo, discutimos las principales APIs de criptografía en .NET:

- API de Protección de Datos de Windows (DPAPI)
- Hashing
- Cifrado simétrico
- Cifrado y firma de clave pública

Los tipos cubiertos en este capítulo están definidos en los siguientes espacios de nombres:

```
System.Security;
System.Security.Cryptography;
```

Resumen

La Tabla 20-1 resume las opciones de criptografía en .NET. En las secciones restantes, exploraremos cada una de ellas.

Option	Keys to manage	Speed	Strength	Notes
`File.Encrypt`	0	Fast	Depends on user's password	Protects files transparently with filesystem support. A key is derived implicitly from the logged-in user's credentials. Windows only.
Windows Data Protection	0	Fast	Depends on user's password	Encrypts and decrypts byte arrays using an implicitly derived key.
Hashing	0	Fast	High	One-way (irreversible) transformation. Used for storing passwords, comparing files, and checking for data corruption.

Option	Keys to manage	Speed	Strength	Notes
Symmetric Encryption	1	Fast	High	For general-purpose encryption/decryption. The same key encrypts and decrypts. Can be used to secure messages in transit.
Public Key Encryption	2	Slow	High	Encryption and decryption use different keys. Used for exchanging a symmetric key in message transmission and for digitally signing files.

.NET también proporciona soporte más especializado para crear y validar firmas basadas en XML en System.Security.Cryptography.Xml y tipos para trabajar con certificados digitales en System.Security.Cryptography.X509Certificates.

Protección de Datos de Windows



La Protección de Datos de Windows está disponible solo en Windows y arroja una PlatformNotSupportedException en otros sistemas operativos

En la sección "Operaciones de archivos y directorios" en la página 699, describimos cómo se puede utilizar File.Encrypt para solicitar al sistema operativo que cifre transparentemente un archivo:

```
File.WriteAllText ("myfile.txt", "");
File.Encrypt ("myfile.txt");
File.AppendAllText ("myfile.txt", "sensitive data");
```

El cifrado en este caso utiliza una clave derivada de la contraseña del usuario que ha iniciado sesión. Puede utilizar esta misma clave derivada implícitamente para cifrar una matriz de bytes con la API de Protección de Datos de Windows (DPAPI). La DPAPI se expone a través de la clase ProtectedData, que es un tipo simple con dos métodos estáticos:

```
public static byte[] Protect
  (byte[] userData, byte[] optionalEntropy, DataProtectionScope scope);

public static byte[] Unprotect
  (byte[] encryptedData, byte[] optionalEntropy, DataProtectionScope scope);
```

Cualquier valor que incluya en optionalEntropy se agrega a la clave, lo que aumenta su seguridad. El argumento DataProtectionScope enum permite dos opciones: CurrentUser (Usuario actual) o LocalMachine (Máquina local). Con CurrentUser, se deriva una clave de las credenciales del usuario que ha iniciado sesión; con LocalMachine, se utiliza una clave a nivel de máquina, común a todos los usuarios. Esto significa que con el ámbito CurrentUser, los datos cifrados por un usuario no pueden ser descifrados por otro. Una clave LocalMachine proporciona menos protección pero funciona en un servicio de Windows o en un programa que necesita operar bajo varias cuentas.

Aquí tienes una demostración simple de cifrado y descifrado:

```
byte[] original = {1, 2, 3, 4, 5};
DataProtectionScope scope = DataProtectionScope.CurrentUser;

byte[] encrypted = ProtectedData.Protect (original, null, scope);
byte[] decrypted = ProtectedData.Unprotect (encrypted, null, scope);
// decrypted is now {1, 2, 3, 4, 5}
```

La Protección de Datos de Windows proporciona una seguridad moderada contra un atacante con acceso completo al equipo, dependiendo de la fortaleza de la contraseña del usuario. Con el ámbito LocalMachine, solo es efectiva contra aquellos con acceso físico y electrónico restringido.

Hashing

El algoritmo de hash condensa un número potencialmente grande de bytes en un hash de longitud fija más pequeño. Los algoritmos de hash están diseñados de tal manera que un cambio de un solo bit en cualquier parte de los datos fuente resulta en un hash significativamente diferente. Esto lo hace adecuado para comparar archivos o detectar corrupción accidental (o maliciosa) en un archivo o flujo de datos.

El hash también actúa como una forma de encriptación unidireccional, ya que es difícil o imposible convertir un hash de vuelta a los datos originales. Esto lo hace ideal para almacenar contraseñas en una base de datos, ya que si tu base de datos se ve comprometida, no quieres que el atacante acceda a contraseñas en texto plano. Para autenticar, simplemente calculas el hash de lo que el usuario ingresa y lo comparas con el hash almacenado en la base de datos.

Para realizar un hash, puedes llamar a ComputeHash en una de las subclases de HashAlgorithm, como SHA1 o SHA256:

```
byte[] hash;
using (Stream fs = File.OpenRead ("checkme.doc"))
  hash = SHA1.Create().ComputeHash (fs);   // SHA1 hash is 20 bytes long
```

El método ComputeHash también acepta un arreglo de bytes, lo cual es conveniente para hacer hash de contraseñas (describiremos una técnica más segura en "Hashing Passwords" en la página 854):

```
byte[] data = System.Text.Encoding.UTF8.GetBytes ("stRhong%pword");
byte[] hash = SHA256.Create().ComputeHash (data);
```



El método GetBytes en un objeto Encoding convierte una cadena de texto en un arreglo de bytes; el método GetString lo convierte de vuelta. Sin embargo, un objeto Encoding no puede convertir un arreglo de bytes encriptado o hasheado a una cadena de texto, ya que los datos enmascarados suelen violar las reglas de codificación de texto. En su lugar, utiliza Convert.ToBase64String y Convert.FromBase64String: estos métodos convierten entre cualquier arreglo de bytes y una cadena legal (amigable para XML o JSON).

Algoritmos de hash en .NET

SHA1 y SHA256 son dos de los subtipos de HashAlgorithm proporcionados por .NET. Aquí están todos los algoritmos principales, en orden ascendente de seguridad (y longitud de hash, en bytes):

```
MD5(16) → SHA1(20) → SHA256(32) → SHA384(48) → SHA512(64)
```

MD5 y SHA1 son actualmente los algoritmos más rápidos, aunque los demás algoritmos no son más de aproximadamente dos veces más lentos en sus implementaciones actuales. Como referencia, puedes esperar un rendimiento de más de 100 MB por segundo con cualquiera de estos algoritmos en los equipos de escritorio o servidores típicos de hoy en día. Los hashes más largos disminuyen la posibilidad de colisión (dos archivos distintos que producen el mismo hash).



Utiliza al menos SHA256 al hacer hash de contraseñas u otros datos sensibles de seguridad. MD5 y SHA1 se consideran inseguros para este propósito y solo son adecuados para protegerse contra la corrupción accidental, no la manipulación deliberada.

Hashing de contraseñas

Los algoritmos SHA más largos son adecuados como base para el hashing de contraseñas, si aplicas una política de contraseñas fuerte para mitigar un ataque de diccionario, una estrategia mediante la cual un atacante construye una tabla de búsqueda de contraseñas hasheando cada palabra en un diccionario.

Una técnica estándar al hacer hash de contraseñas es incorporar "salt" (sal) —una larga serie de bytes que inicialmente obtienes mediante un generador de números aleatorios y luego combinas con cada contraseña antes de hacer el hash. Esto frustra a los hackers de dos formas:

- También deben conocer los bytes de sal.
- No pueden usar tablas arcoíris (bases de datos precomputadas y disponibles públicamente de contraseñas y sus hashcodes), aunque un ataque de diccionario aún podría ser posible con suficiente poder de cómputo.

Puedes fortalecer aún más la seguridad al "estirar" tus hashes de contraseñas, es decir, volver a hacer el hash repetidamente para obtener secuencias de bytes más intensivas computacionalmente. Si vuelves a hacer el hash 100 veces, un ataque de diccionario que de otra manera tomaría un mes tomaría ocho años. Las clases KeyDerivation, Rfc2898DeriveBytes y PasswordDeriveBytes realizan exactamente este tipo de estiramiento y también permiten el uso conveniente de sal. De estas, KeyDerivation.Pbkdf2 ofrece el mejor hashing:

```
byte[] encrypted = KeyDerivation.Pbkdf2 (
    password: "stRhong%pword",
    salt: Encoding.UTF8.GetBytes ("j78Y#p)/saREN!y3@"),
    prf: KeyDerivationPrf.HMACSHA512,
    iterationCount: 100,
    numBytesRequested: 64);
```



KeyDerivation.Pbkdf2 requiere el paquete NuGet Microsoft.AspNetCore.Cryptography.KeyDerivation. Aunque está en el espacio de nombres de ASP.NET Core, cualquier aplicación .NET puede usarlo.

Cifrado simétrico

El cifrado simétrico utiliza la misma clave tanto para el cifrado como para el descifrado. La BCL de .NET proporciona cuatro algoritmos simétricos, de los cuales Rijndael (pronunciado "Rhine Dahl" o "Rain Doll") es el principal; los otros algoritmos están destinados principalmente a la compatibilidad con aplicaciones más antiguas. Rijndael es rápido y seguro, y tiene dos implementaciones:

- La clase Rijndael
- La clase Aes

Ambas son casi idénticas, excepto que Aes no te permite debilitar el cifrado cambiando el tamaño del bloque. Aes es recomendado por el equipo de seguridad del CLR. Rijndael y Aes permiten claves simétricas de longitud de 16, 24 o 32 bytes: todas se consideran seguras actualmente. Aquí tienes un ejemplo de cómo cifrar una serie de bytes mientras se escriben en un archivo, utilizando una clave de 16 bytes:

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};
byte[] iv  = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};

byte[] data = { 1, 2, 3, 4, 5 };   // This is what we're encrypting.

using (SymmetricAlgorithm algorithm = Aes.Create())
using (ICryptoTransform encryptor = algorithm.CreateEncryptor (key, iv))
using (Stream f = File.Create ("encrypted.bin"))
using (Stream c = new CryptoStream (f, encryptor, CryptoStreamMode.Write))
  c.Write (data, 0, data.Length);
```

El siguiente código descifra el archivo:

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};
byte[] iv  = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};

byte[] decrypted = new byte[5];

using (SymmetricAlgorithm algorithm = Aes.Create())
using (ICryptoTransform decryptor = algorithm.CreateDecryptor (key, iv))
using (Stream f = File.OpenRead ("encrypted.bin"))
using (Stream c = new CryptoStream (f, decryptor, CryptoStreamMode.Read))
  for (int b; (b = c.ReadByte()) > -1;)
    Console.Write (b + " ");                          // 1 2 3 4 5
```

En este ejemplo, creamos una clave de 16 bytes elegidos al azar. Si se utiliza la clave incorrecta en el descifrado, CryptoStream lanzará una CryptographicException. Capturar esta excepción es la única forma de probar si una clave es correcta.

Así como una clave, creamos un IV (Initialization Vector o Vector de Inicialización). Esta secuencia de 16 bytes forma parte del cifrado, al igual que la clave, pero no se considera secreta. Si estás transmitiendo un mensaje cifrado, enviarías el IV en texto plano (quizás en un encabezado del mensaje) y luego lo cambiarías con cada mensaje. Esto haría que cada mensaje cifrado sea irreconocible en comparación con cualquier mensaje anterior, incluso si sus versiones sin cifrar son similares o idénticas.



Si no necesitas o no deseas la protección de un IV, puedes evitarlo utilizando el mismo valor de 16 bytes tanto para la clave como para el IV. Sin embargo, el uso de múltiples mensajes con el mismo IV debilita el cifrado e incluso podría hacer posible descifrarlo.

El trabajo de criptografía se divide entre las clases. Aes es el matemático; aplica el algoritmo de cifrado, junto con sus transformaciones de cifrado y descifrado. CryptoStream es el fontanero; se encarga de la canalización del flujo. Puedes reemplazar Aes con un algoritmo simétrico diferente y seguir utilizando CryptoStream.

CryptoStream es bidireccional, lo que significa que puedes leer o escribir en el flujo dependiendo de si eliges CryptoStreamMode.Read o CryptoStreamMode.Write. Tanto los transformadores de cifrado como los de descifrado son hábiles en lectura y escritura, lo que da lugar a cuatro combinaciones. La elección puede hacer que te quedes mirando una pantalla en blanco durante un tiempo. Puede ser útil modelar la lectura como "extracción" y la escritura como "inserción". Si tienes dudas, comienza con Write para el cifrado y Read para el descifrado, ya que esto suele ser lo más natural.

Para generar una clave o un IV aleatorio, utiliza la clase RandomNumberGenerator en System.Cryptography. Los números que produce son verdaderamente impredecibles o criptográficamente fuertes (la clase System.Random no ofrece la misma garantía). Aquí tienes un ejemplo:

```
byte[] key = new byte [16];
byte[] iv  = new byte [16];
RandomNumberGenerator rand = RandomNumberGenerator.Create();
rand.GetBytes (key);
rand.GetBytes (iv);
```

O, a partir de .NET 6:

```
byte[] key = RandomNumberGenerator.GetBytes (16);
byte[] iv = RandomNumberGenerator.GetBytes (16);
```

Si no especificas una clave y un IV, se generarán automáticamente valores aleatorios criptográficamente fuertes. Puedes obtener estos valores a través de las propiedades Key e IV del objeto Aes.

Cifrado en memoria

A partir de .NET 6, puedes utilizar los métodos EncryptCbc y DecryptCbc para simplificar el proceso de cifrado y descifrado de matrices de bytes:

```
public static byte[] Encrypt (byte[] data, byte[] key, byte[] iv)
{
  using Aes algorithm = Aes.Create();
```

```
  algorithm.Key = key;
  return algorithm.EncryptCbc (data, iv);
}

public static byte[] Decrypt (byte[] data, byte[] key, byte[] iv)
{
  using Aes algorithm = Aes.Create();
  algorithm.Key = key;
  return algorithm.DecryptCbc (data, iv);
}
```

Aquí tienes un código equivalente que funciona en todas las versiones de .NET:

```
public static byte[] Encrypt (byte[] data, byte[] key, byte[] iv)
{
  using (Aes algorithm = Aes.Create())
  using (ICryptoTransform encryptor = algorithm.CreateEncryptor (key, iv))
    return Crypt (data, encryptor);
}

public static byte[] Decrypt (byte[] data, byte[] key, byte[] iv)
{
  using (Aes algorithm = Aes.Create())
  using (ICryptoTransform decryptor = algorithm.CreateDecryptor (key, iv))
    return Crypt (data, decryptor);
}

static byte[] Crypt (byte[] data, ICryptoTransform cryptor)
{
  MemoryStream m = new MemoryStream();
  using (Stream c = new CryptoStream (m, cryptor, CryptoStreamMode.Write))
    c.Write (data, 0, data.Length);
  return m.ToArray();
}
```

Aquí, el modo CryptoStreamMode.Write funciona mejor tanto para el cifrado como para el descifrado, ya que en ambos casos estamos "empujando" los datos hacia una nueva secuencia de memoria.

Aquí tienes las sobrecargas que aceptan y devuelven cadenas de texto:

```
  public static string Encrypt (string data, byte[] key, byte[] iv)
  {
    return Convert.ToBase64String (
      Encrypt (Encoding.UTF8.GetBytes (data), key, iv));
  }

  public static string Decrypt (string data, byte[] key, byte[] iv)
  {
    return Encoding.UTF8.GetString (
      Decrypt (Convert.FromBase64String (data), key, iv));
  }
```

Lo siguiente demuestra cómo se utilizan:

```
byte[] key = new byte[16];
byte[] iv = new byte[16];
```

```
var cryptoRng = RandomNumberGenerator.Create();
cryptoRng.GetBytes (key);
cryptoRng.GetBytes (iv);

string encrypted = Encrypt ("Yeah!", key, iv);
Console.WriteLine (encrypted);                // R1/5gYvcxyR2vzPjnT7yaQ==

string decrypted = Decrypt (encrypted, key, iv);
Console.WriteLine (decrypted);                // Yeah!
```

Chaining Encryption Streams (Cadenas de transmisión de cifrado)

CryptoStream es un decorador, lo que significa que se puede encadenar con otros flujos de datos. En el siguiente ejemplo, escribimos texto cifrado comprimido en un archivo y luego lo leemos de nuevo:

```
byte[] key = new byte [16];
byte[] iv = new byte [16];

var cryptoRng = RandomNumberGenerator.Create();
cryptoRng.GetBytes (key);
cryptoRng.GetBytes (iv);

using (Aes algorithm = Aes.Create())
{
  using (ICryptoTransform encryptor = algorithm.CreateEncryptor(key, iv))
  using (Stream f = File.Create ("serious.bin"))
  using (Stream c = new CryptoStream (f, encryptor, CryptoStreamMode.Write))
  using (Stream d = new DeflateStream (c, CompressionMode.Compress))
  using (StreamWriter w = new StreamWriter (d))
    await w.WriteLineAsync ("Small and secure!");

  using (ICryptoTransform decryptor = algorithm.CreateDecryptor(key, iv))
  using (Stream f = File.OpenRead ("serious.bin"))
  using (Stream c = new CryptoStream (f, decryptor, CryptoStreamMode.Read))
  using (Stream d = new DeflateStream (c, CompressionMode.Decompress))
  using (StreamReader r = new StreamReader (d))
    Console.WriteLine (await r.ReadLineAsync());     // Small and secure!
}
```

(Como toque final, hacemos que nuestro programa sea asíncrono llamando a WriteLineAsync y ReadLineAsync y esperando el resultado).

En este ejemplo, todas las variables de una sola letra forman parte de una cadena. Los elementos matemáticos: algoritmo, encryptor y decryptor, están presentes para ayudar a CryptoStream en el proceso de cifrado, como se ilustra en la Figura 20-1.

El encadenamiento de flujos de esta manera requiere poca memoria, independientemente del tamaño final del flujo de datos.

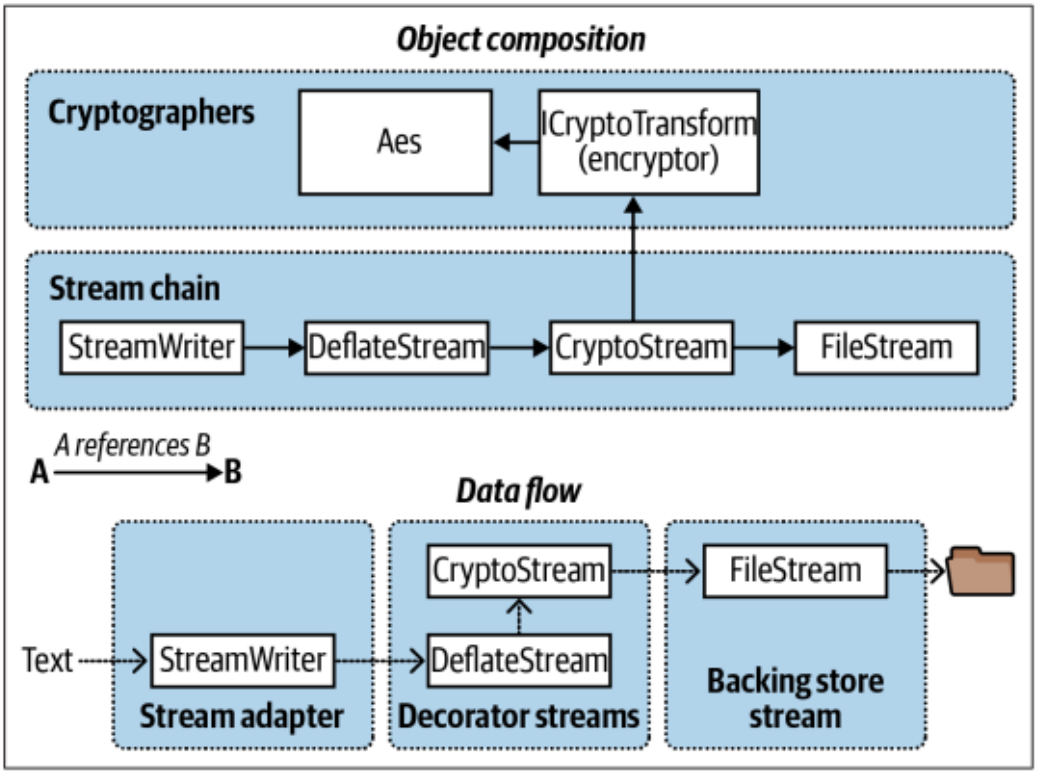


Figure 20-1. Chaining encryption and compression streams

Eliminando Objetos de Cifrado de forma segura (Disposing Encryption Objects)

Al disponer de un CryptoStream, nos aseguramos de que su caché interna de datos se vacíe en el flujo subyacente. El almacenamiento en caché interno es necesario para los algoritmos de cifrado, ya que procesan los datos en bloques en lugar de un byte a la vez.

CryptoStream es inusual en el sentido de que su método Flush no hace nada. Para vaciar un flujo (sin disponerlo), debes llamar a FlushFinalBlock. A diferencia de Flush, solo puedes llamar a FlushFinalBlock una vez y luego no se pueden escribir más datos.

También disponemos de los elementos matemáticos: el objeto Aes y los objetos ICryptoTransform (encryptor y decryptor). Cuando se disponen los objetos de transformación Rijndael, se eliminan la clave simétrica y los datos relacionados de la memoria, evitando su posterior descubrimiento por parte de otros software que se ejecute en la computadora (estamos hablando de malware). No puedes confiar en el recolector de basura para esta tarea, ya que solo marca secciones de memoria como disponibles; no escribe ceros en cada byte.

La forma más sencilla de disponer de un objeto Aes fuera de una instrucción using es llamar a Clear. Su método Dispose está oculto mediante una implementación explícita (para señalar su inusual semántica de disposición, mediante la cual se borra la memoria en lugar de liberar recursos no administrados).



Puedes reducir aún más la vulnerabilidad de tu aplicación frente a la filtración de secretos a través de la memoria liberada siguiendo estos pasos:

- Evita utilizar cadenas de texto para información de seguridad (al ser inmutables, el valor de una cadena de texto nunca se puede borrar una vez creada).
- Sobrescribe los buffers tan pronto como ya no sean necesarios (por ejemplo, llamando a Array.Clear en un arreglo de bytes).

Gestión de Claves

La gestión de claves es un elemento crítico en la seguridad: si tus claves se exponen, también se expone tu información. Debes considerar quién debe tener acceso a las claves y cómo realizar copias de seguridad en caso de falla de hardware, almacenándolas de manera que evite el acceso no autorizado.

No es recomendable codificar las claves de cifrado directamente en el código, ya que existen herramientas populares para descompilar ensamblados sin requerir mucha experiencia. Una opción mejor (en Windows) es generar una clave aleatoria para cada instalación y almacenarla de forma segura mediante Windows Data Protection.

Para aplicaciones implementadas en la nube, Microsoft Azure y Amazon Web Services (AWS) ofrecen sistemas de gestión de claves con características adicionales que pueden ser útiles en un entorno empresarial, como registros de auditoría.

Si estás cifrando un flujo de mensajes, la encriptación de clave pública sigue siendo la mejor opción.

Cifrado y Firma con Clave Pública

La criptografía de clave pública es asimétrica, lo que significa que el cifrado y el descifrado utilizan claves diferentes.

A diferencia del cifrado simétrico, para el cual cualquier serie arbitraria de bytes de la longitud adecuada puede servir como clave, la criptografía asimétrica requiere pares de claves especialmente diseñadas. Un par de claves contiene una clave pública y una clave privada que funcionan juntas de la siguiente manera:

- La clave pública cifra los mensajes.
- La clave privada descifra los mensajes.

La parte que "crea" un par de claves mantiene la clave privada en secreto mientras distribuye libremente la clave pública. Una característica especial de este tipo de criptografía es que no es posible calcular una clave privada a partir de una clave pública. Por lo tanto, si se pierde la clave privada, no se puede recuperar los datos cifrados; por el contrario, si se filtra una clave privada, el sistema de encriptación se vuelve inútil.

Un intercambio de claves públicas permite que dos computadoras se comuniquen de forma segura a través de una red pública, sin contacto previo y sin ningún secreto compartido existente. Para entender cómo funciona esto, supongamos que la computadora de origen (Origin) desea enviar un mensaje confidencial a la computadora de destino (Target):

1. Target genera un par de claves pública/privada y luego envía su clave pública a Origin.

2. Origin cifra el mensaje confidencial utilizando la clave pública de Target y luego lo envía a Target.

3. Target descifra el mensaje confidencial utilizando su clave privada.

Un espía verá lo siguiente:

- La clave pública de Target.
- El mensaje secreto cifrado con la clave pública de Target.

Pero sin la clave privada de Target, el mensaje no puede ser descifrado.



Esto no protege contra un ataque de intermediario (man-in-the-middle): en otras palabras, Origin no puede saber si Target es una parte maliciosa. Para autenticar al destinatario, el remitente necesita conocer previamente la clave pública del destinatario o poder validar su clave a través de un certificado digital.

Dado que el cifrado de clave pública es relativamente lento y tiene un límite en el tamaño del mensaje, el mensaje secreto enviado de Origin a Target generalmente contiene una clave nueva para el cifrado simétrico posterior. Esto permite abandonar el cifrado de clave pública durante el resto de la sesión, a favor de un algoritmo simétrico capaz de manejar mensajes más grandes. Este protocolo es especialmente seguro si se genera un par de claves pública/privada nuevo para cada sesión, ya que no se necesitan almacenar claves en ninguna de las computadoras.



Los algoritmos de cifrado de clave pública se basan en que el mensaje sea más pequeño que la clave. Esto los hace adecuados solo para cifrar cantidades pequeñas de datos, como una clave para el cifrado simétrico posterior. Si intentas cifrar un mensaje mucho más grande que la mitad del tamaño de la clave, el proveedor lanzará una excepción.

La clase RSA

.NET proporciona varios algoritmos asimétricos, de los cuales RSA es el más popular. Así es como se cifra y descifra con RSA:

```
byte[] data = { 1, 2, 3, 4, 5 };   // This is what we're encrypting.

using (var rsa = new RSACryptoServiceProvider())
{
  byte[] encrypted = rsa.Encrypt (data, true);
  byte[] decrypted = rsa.Decrypt (encrypted, true);
}
```

Dado que no especificamos una clave pública o privada, el proveedor criptográfico generó automáticamente un par de claves utilizando la longitud predeterminada de 1.024 bits. Puedes

solicitar claves más largas en incrementos de ocho bytes a través del constructor. Para aplicaciones críticas de seguridad, es prudente solicitar claves de 2.048 bits:

```
var rsa = new RSACryptoServiceProvider (2048);
```

Generar un par de claves es intensivo en cómputo y puede llevar aproximadamente 100 ms. Por esta razón, la implementación de RSA retrasa este proceso hasta que se necesite realmente una clave, como al llamar a `Encrypt`. Esto te brinda la oportunidad de cargar una clave existente, o un par de claves si ya existe.

Los métodos `ImportCspBlob` y `ExportCspBlob` permiten cargar y guardar claves en formato de matriz de bytes. `FromXmlString` y `ToXmlString` realizan la misma tarea en un formato de cadena, donde la cadena contiene un fragmento XML. Un indicador booleano te permite especificar si se incluye la clave privada al guardarla. A continuación, se muestra cómo generar un par de claves y guardarlo en el disco:

```
using (var rsa = new RSACryptoServiceProvider())
{
  File.WriteAllText ("PublicKeyOnly.xml", rsa.ToXmlString (false));
  File.WriteAllText ("PublicPrivate.xml", rsa.ToXmlString (true));
}
```

Dado que no proporcionamos claves existentes, `ToXmlString` forzó la generación de un nuevo par de claves (en la primera llamada). En el siguiente ejemplo, leemos estas claves desde el disco y las utilizamos para cifrar y descifrar un mensaje:

```
byte[] data = Encoding.UTF8.GetBytes ("Message to encrypt");

string publicKeyOnly = File.ReadAllText ("PublicKeyOnly.xml");
string publicPrivate = File.ReadAllText ("PublicPrivate.xml");

byte[] encrypted, decrypted;

using (var rsaPublicOnly = new RSACryptoServiceProvider())
{
  rsaPublicOnly.FromXmlString (publicKeyOnly);
  encrypted = rsaPublicOnly.Encrypt (data, true);

  // The next line would throw an exception because you need the private
  // key in order to decrypt:
  // decrypted = rsaPublicOnly.Decrypt (encrypted, true);
}

using (var rsaPublicPrivate = new RSACryptoServiceProvider())
{
  // With the private key we can successfully decrypt:
  rsaPublicPrivate.FromXmlString (publicPrivate);
  decrypted = rsaPublicPrivate.Decrypt (encrypted, true);
}
```

Digital Signing (Firma digital)

También puedes utilizar algoritmos de clave pública para firmar digitalmente mensajes o documentos. Una firma es similar a un hash, excepto que su producción requiere una clave

privada y, por lo tanto, no puede ser falsificada. La clave pública se utiliza para verificar la firma. Aquí tienes un ejemplo:

```
byte[] data = Encoding.UTF8.GetBytes ("Message to sign");
byte[] publicKey;
byte[] signature;
object hasher = SHA1.Create();         // Our chosen hashing algorithm.

// Generate a new key pair, then sign the data with it:
using (var publicPrivate = new RSACryptoServiceProvider())
{
  signature = publicPrivate.SignData (data, hasher);
  publicKey = publicPrivate.ExportCspBlob (false);    // get public key
}

// Create a fresh RSA using just the public key, then test the signature.
using (var publicOnly = new RSACryptoServiceProvider())
{
  publicOnly.ImportCspBlob (publicKey);
  Console.Write (publicOnly.VerifyData (data, hasher, signature)); // True

  // Let's now tamper with the data and recheck the signature:
  data[0] = 0;
  Console.Write (publicOnly.VerifyData (data, hasher, signature)); // False

  // The following throws an exception as we're lacking a private key:
  signature = publicOnly.SignData (data, hasher);
}
```

La firma funciona primero realizando un hash de los datos y luego aplicando el algoritmo asimétrico al hash resultante. Debido a que los hashes tienen un tamaño fijo y pequeño, los documentos grandes pueden ser firmados relativamente rápido (el cifrado de clave pública requiere mucho más CPU que el hashing). Si lo deseas, puedes realizar el hash tú mismo y luego llamar a `SignHash` en lugar de `SignData`:

```
using (var rsa = new RSACryptoServiceProvider())
{
  byte[] hash = SHA1.Create().ComputeHash (data);
  signature = rsa.SignHash (hash, CryptoConfig.MapNameToOID ("SHA1"));
  ...
}
```

`SignHash` aún necesita saber qué algoritmo de hash utilizaste; `CryptoConfig.MapNameToOID` proporciona esta información en el formato correcto a partir de un nombre amigable como "SHA1".

`RSACryptoServiceProvider` produce firmas cuyo tamaño coincide con el de la clave. Actualmente, ningún algoritmo principal produce firmas seguras significativamente más pequeñas que 128 bytes (adecuadas para códigos de activación de productos, por ejemplo).



Para que la firma sea efectiva, el destinatario debe conocer y confiar en la clave pública del remitente. Esto puede ocurrir a través de comunicación previa, preconfiguración o un certificado del sitio. Un certificado del sitio es un registro electrónico de la clave pública y el nombre del originador, firmado por una autoridad de confianza independiente. El espacio de nombres System.Security.Cryptography.X509Certificates define los tipos para trabajar con certificados.

21 Programación avanzada de subprocesos

Comenzamos el Capítulo 14 con los conceptos básicos de la programación de subprocesos como precursor de las tareas y la asincronía. Específicamente, mostramos cómo iniciar y configurar un subproceso, y cubrimos conceptos esenciales como la agrupación de subprocesos, el bloqueo, el giro y los contextos de sincronización. También introdujimos el bloqueo y la seguridad de subprocesos, y demostramos la construcción de señalización más simple, ManualResetEvent.

Este capítulo retoma el tema de la programación de subprocesos donde lo dejamos en el Capítulo 14. En las primeras tres secciones, ampliamos la sincronización, el bloqueo y la seguridad de subprocesos en mayor detalle. Luego cubrimos:

- Bloqueo no exclusivo (Semaphore y bloqueo de lector/escritor)
- Todas las construcciones de señalización (AutoResetEvent, ManualResetEvent, CountdownEvent y Barrier)
- Inicialización diferida (Lazy<T> y LazyInitializer)
- Almacenamiento local de subprocesos (ThreadStaticAttribute, ThreadLocal<T> y GetData/SetData)
- Temporizadores

La programación de subprocesos es un tema tan amplio que hemos incluido material adicional en línea para completar el panorama. Visita http://albahari.com/threading para obtener una discusión sobre los siguientes temas más arcanos:

- Monitor.Wait y Monitor.Pulse para escenarios de señalización especializados
- Técnicas de sincronización sin bloqueo para micro-optimización (Interlocked, barreras de memoria, volatile)
- SpinLock y SpinWait para escenarios de alta concurrencia.

Resumen de Sincronización

La sincronización es el acto de coordinar acciones concurrentes para obtener un resultado predecible. La sincronización es particularmente importante cuando varios subprocesos acceden a los mismos datos; es sorprendentemente fácil encontrarse con problemas en esta área.

Las herramientas de sincronización más simples y útiles son, sin duda, las continuaciones y los combinadores de tareas descritos en el Capítulo 14. Al formular programas concurrentes en operaciones asíncronas concatenadas con continuaciones y combinadores, se reduce la necesidad de bloqueo y señalización. Sin embargo, aún hay momentos en los que entran en juego las construcciones de nivel inferior.

Las construcciones de sincronización se pueden dividir en tres categorías:

Bloqueo exclusivo

Las construcciones de bloqueo exclusivo permiten que solo un subproceso realice una actividad o ejecute una sección de código a la vez. Su propósito principal es permitir que los subprocesos accedan al estado de escritura compartido sin interferir entre sí. Las construcciones de bloqueo exclusivo son lock, Mutex y SpinLock.

Bloqueo no exclusivo

El bloqueo no exclusivo le permite limitar la concurrencia. Las construcciones de bloqueo no exclusivo son Semaphore(Slim) y ReaderWriterLock(Slim).

Señalización

Estas permiten que un subproceso se bloquee hasta recibir una o más notificaciones de otro(s) subproceso(s). Las construcciones de señalización incluyen ManualResetEvent(Slim), AutoResetEvent, CountdownEvent y Barrier. Las primeras tres se denominan manejadores de espera de eventos.

También es posible (y complicado) realizar ciertas operaciones concurrentes en estado compartido sin bloqueo mediante el uso de construcciones de sincronización sin bloqueo. Estas son Thread.MemoryBarrier, Thread.VolatileRead, Thread.VolatileWrite, la palabra clave volatile y la clase Interlocked. Cubrimos este tema en línea, junto con los métodos Wait/Pulse de Monitor, que se pueden utilizar para escribir lógica de señalización personalizada.

Bloqueo exclusivo

Existen tres construcciones de bloqueo exclusivo: la declaración lock, Mutex y SpinLock. La construcción lock es la más conveniente y ampliamente utilizada, mientras que las otras dos se centran en escenarios específicos:

• Mutex te permite abarcar múltiples procesos (bloqueos en todo el equipo).

• SpinLock implementa una microoptimización que puede reducir los cambios de contexto en escenarios de alta concurrencia (ver http://albahari.com/threading).

La declaración lock

Para ilustrar la necesidad de bloqueo, considera la siguiente clase:

```
class ThreadUnsafe
{
  static int _val1 = 1, _val2 = 1;

  static void Go()
  {
    if (_val2 != 0) Console.WriteLine (_val1 / _val2);
    _val2 = 0;
  }
}
```

Esta clase no es segura para subprocesos: si Go fuera llamado por dos subprocesos simultáneamente, sería posible obtener un error de división por cero porque _val2 podría

establecerse en cero en un subproceso justo cuando el otro subproceso estaba entre la ejecución de la declaración if y Console.WriteLine. Así es como el bloqueo resuelve el problema:

```
class ThreadSafe
{
  static readonly object _locker = new object();
  static int _val1 = 1, _val2 = 1;

  static void Go()
  {
    lock (_locker)
    {
      if (_val2 != 0) Console.WriteLine (_val1 / _val2);
      _val2 = 0;
    }
  }
}
```

Solo un subproceso puede bloquear el objeto de sincronización (en este caso, _locker) a la vez, y cualquier subproceso en disputa se bloquea hasta que se libera el bloqueo. Si más de un subproceso compite por el bloqueo, se colocan en una "cola de espera" y se les otorga el bloqueo en orden de llegada. A veces se dice que los bloqueos exclusivos hacen cumplir el acceso serializado a lo que está protegido por el bloqueo porque el acceso de un subproceso no puede superponerse con el de otro. En este caso, estamos protegiendo la lógica dentro del método Go, así como los campos _val1 y _val2.

Monitor.Enter y Monitor.Exit

La declaración lock de C# es en realidad una abreviatura sintáctica de una llamada a los métodos Monitor.Enter y Monitor.Exit, con un bloque try/finally. Aquí está (una versión simplificada de) lo que realmente sucede dentro del método Go del ejemplo anterior:

```
Monitor.Enter (_locker);
try
{
  if (_val2 != 0) Console.WriteLine (_val1 / _val2);
  _val2 = 0;
}
finally { Monitor.Exit (_locker); }
```

Llamar a Monitor.Exit sin haber llamado previamente a Monitor.Enter en el mismo objeto genera una excepción.

Las sobrecargas de lockTaken

El código que acabamos de demostrar tiene una vulnerabilidad sutil. Considera el evento (improbable) de que se lance una excepción entre la llamada a Monitor.Enter y el bloque try (debido, tal vez, a una OutOfMemoryException o, en .NET Framework, si el subproceso es abortado). En tal escenario, es posible que el bloqueo se haya tomado o no. Si el bloqueo se ha tomado, no se liberará, porque nunca entraremos en el bloque try/finally. Esto dará como

resultado un bloqueo sin liberar. Para evitar este peligro, Monitor.Enter define la siguiente sobrecarga:

```
public static void Enter (object obj, ref bool lockTaken);
```

lockTaken es false después de llamar a este método si (y solo si) el método Enter genera una excepción y el bloqueo no se tomó.

Aquí está el patrón de uso más robusto (que es exactamente cómo C# traduce una declaración lock):

```
bool lockTaken = false;
try
{
  Monitor.Enter (_locker, ref lockTaken);
  // Do your stuff...
}
finally { if (lockTaken) Monitor.Exit (_locker); }
```

TryEnter

Monitor también proporciona un método TryEnter que permite especificar un tiempo de espera, ya sea en milisegundos o como un TimeSpan. El método devuelve true si se obtuvo un bloqueo, o false si no se obtuvo un bloqueo porque el método expiró. TryEnter también puede ser llamado sin argumentos, lo que "prueba" el bloqueo y expira inmediatamente si no se puede obtener el bloqueo de inmediato. Al igual que el método Enter, TryEnter tiene sobrecargas para aceptar un argumento lockTaken.

Elección del Objeto de Sincronización (Choosing the Synchronization Object)

Puedes utilizar cualquier objeto visible para cada uno de los hilos participantes como objeto de sincronización, sujeto a una regla importante: debe ser un tipo de referencia. El objeto de sincronización suele ser privado (porque esto ayuda a encapsular la lógica de bloqueo) y normalmente es un campo de instancia o estático. El objeto de sincronización puede funcionar también como el objeto que está protegiendo, como lo hace el campo `_list` en el siguiente ejemplo:

```
class ThreadSafe
{
  List <string> _list = new List <string>();

  void Test()
  {
    lock (_list)
    {
      _list.Add ("Item 1");
      ...
```

Un campo dedicado específicamente para fines de bloqueo (como `_locker` en el ejemplo anterior) permite un control preciso sobre el alcance y la granularidad del bloqueo. También puedes usar el objeto contenedor (`this`) como objeto de sincronización:

```
lock (this) { ... }
```

Incluso puedes usar el tipo de objeto como objeto de sincronización:

```
lock (typeof (Widget)) { ... }    // For protecting access to statics
```

La desventaja de bloquear de esta manera es que no estás encapsulando la lógica de bloqueo, por lo que se vuelve más difícil evitar el bloqueo muerto y el bloqueo excesivo. También puedes bloquear variables locales capturadas por expresiones lambda o métodos anónimos.



El bloqueo no restringe el acceso al objeto de sincronización en sí de ninguna manera. En otras palabras, `x.ToString()` no se bloqueará porque otro hilo haya llamado a `lock(x)`. Ambos hilos deben llamar a `lock(x)` para que ocurra el bloqueo.

Cuándo usar el bloqueo

Como regla básica, debes usar bloqueo alrededor de cualquier acceso a un campo compartido que pueda ser modificado. Incluso en el caso más simple, una operación de asignación en un solo campo, debes considerar la sincronización. En la siguiente clase, ni el método `Increment` ni el método `Assign` son seguros para subprocesos:

```
class ThreadUnsafe
{
  static int _x;
  static void Increment() { _x++; }
  static void Assign()    { _x = 123; }
}
```

Aquí tienes versiones seguras para subprocesos de `Increment` y `Assign`:

```
static readonly object _locker = new object();
static int _x;

static void Increment() { lock (_locker) _x++; }
static void Assign()    { lock (_locker) _x = 123; }
```

Sin bloqueos, pueden surgir dos problemas:

- Las operaciones como incrementar una variable (o incluso leer/escribir una variable, bajo ciertas condiciones) no son atómicas.
- El compilador, CLR y el procesador tienen el derecho de reordenar instrucciones y almacenar variables en registros de CPU para mejorar el rendimiento, siempre y cuando estas optimizaciones no cambien el comportamiento de un programa de un solo hilo (o un programa multihilo que utiliza bloqueos).

El bloqueo mitiga el segundo problema porque crea una barrera de memoria antes y después del bloqueo. Una barrera de memoria es una "barrera" a través de la cual los efectos de la reordenación y el almacenamiento en caché no pueden penetrar.



Esto se aplica no solo a los bloqueos, sino a todos los constructos de sincronización. Por lo tanto, si tu uso de un constructo de señalización, por ejemplo, garantiza que solo un hilo lee/escribe una variable a la vez, no necesitas bloquear. Por lo tanto, el siguiente código es seguro para subprocesos sin bloquear alrededor de x:

```
var signal = new ManualResetEvent (false);
int x = 0;
new Thread (() => { x++; signal.Set(); }).Start();
signal.WaitOne();
Console.WriteLine (x);    // 1 (always)
```

En "Sincronización sin bloqueo", explicamos cómo surge esta necesidad y cómo las barreras de memoria y la clase Interlocked pueden proporcionar alternativas al bloqueo en estas situaciones.

Bloqueo y Atomicidad

Si un grupo de variables se lee y se escribe siempre dentro del mismo bloqueo, se puede decir que las variables se leen y se escriben atómicamente. Supongamos que los campos x e y siempre se leen y se asignan dentro de un bloqueo en el objeto locker:

```
lock (locker) { if (x != 0) y /= x; }
```

Podemos decir que x e y se acceden atómicamente porque el bloque de código no puede dividirse o ser interrumpido por las acciones de otro hilo de tal manera que cambie x o y e invalide su resultado. Nunca obtendrás un error de división por cero, siempre y cuando x e y se accedan siempre dentro de este mismo bloqueo exclusivo.



La atomicidad proporcionada por un bloqueo se viola si se lanza una excepción dentro del bloqueo (independientemente de si está involucrado el subprocesamiento múltiple). Por ejemplo, considera lo siguiente:

```
decimal _savingsBalance, _checkBalance;

void Transfer (decimal amount)
{
  lock (_locker)
  {
    _savingsBalance += amount;
    _checkBalance -= amount + GetBankFee();
  }
}
```

Si se lanza una excepción en GetBankFee(), el banco perdería dinero. En este caso, podríamos evitar el problema llamando a GetBankFee antes. Una solución para casos más complejos es implementar una lógica de "reversión" dentro de un bloque catch o finally.

La atomicidad de la instrucción es un concepto diferente, aunque análogo: una instrucción es atómica si se ejecuta indivisiblemente en el procesador subyacente.

Bloqueo anidado

Un hilo puede bloquear repetidamente el mismo objeto de forma anidada (reentrante):

```
lock (locker)
  lock (locker)
    lock (locker)
    {
       // Do something...
    }
```

Alternativamente:

```
Monitor.Enter (locker); Monitor.Enter (locker);  Monitor.Enter (locker);
// Do something...
Monitor.Exit (locker);  Monitor.Exit (locker);   Monitor.Exit (locker);
```

En estos escenarios, el objeto se desbloquea solo cuando se ha salido de la declaración de bloqueo más externa, o se han ejecutado un número correspondiente de declaraciones Monitor.Exit.

El bloqueo anidado es útil cuando un método llama a otro desde dentro de un bloqueo:

```
object locker = new object();

lock (locker)
{
  AnotherMethod();
  // We still have the lock - because locks are reentrant.
}

void AnotherMethod()
{
  lock (locker) { Console.WriteLine ("Another method"); }
}
```

Un hilo solo puede bloquearse en el primer (más externo) bloqueo.

Bloqueos

Un bloqueo ocurre cuando dos hilos esperan cada uno un recurso que posee el otro, por lo que ninguno puede avanzar. La forma más fácil de ilustrar esto es con dos bloqueos:

```
object locker1 = new object();
object locker2 = new object();

new Thread (() => {
                    lock (locker1)
                    {
                      Thread.Sleep (1000);
                      lock (locker2);      // Deadlock
                    }
                  }).Start();
lock (locker2)
{
  Thread.Sleep (1000);
  lock (locker1);                          // Deadlock
}
```

Se pueden crear cadenas de bloqueo más elaboradas con tres o más hilos.



El CLR, en un entorno de hospedaje estándar, no es como SQL Server y no detecta y resuelve automáticamente los bloqueos al finalizar uno de los involucrados. Un bloqueo de subprocesamiento hace que los hilos participantes se bloqueen indefinidamente, a menos que hayas especificado un tiempo de espera para el bloqueo. (Sin embargo, bajo la integración de CLR de SQL, los bloqueos se detectan automáticamente y se produce una excepción [capturable] en uno de los hilos).

Los bloqueos mutuos son uno de los problemas más difíciles en el subprocesamiento, especialmente cuando hay muchos objetos interrelacionados. Fundamentalmente, el problema complicado es que no puedes estar seguro de qué bloqueos ha tomado tu llamador.

Por ejemplo, puedes bloquear el campo privado "a" dentro de tu clase "x", sin saber que tu llamador (o el llamador de tu llamador) ya ha bloqueado el campo "b" dentro de la clase "y". Mientras tanto, otro hilo está haciendo lo contrario, creando un bloqueo mutuo. Irónicamente, el problema se agrava con (buenos) patrones de diseño orientados a objetos, porque dichos patrones crean cadenas de llamadas que no se determinan hasta tiempo de ejecución.

El consejo popular de "bloquear objetos en un orden consistente para prevenir bloqueos mutuos", aunque útil en nuestro ejemplo inicial, es difícil de aplicar al escenario descrito anteriormente. Una estrategia mejor es tener cuidado al bloquear alrededor de llamadas a métodos en objetos que podrían tener referencias de vuelta a tu propio objeto. Además, considera si realmente necesitas bloquear alrededor de llamadas a métodos en otras clases (a menudo sí, como verás en "Bloqueo y seguridad de subprocesos", en la página 874, pero a veces hay otras opciones). Depender más de opciones de sincronización de alto nivel, como las continuaciones/combinadores de tareas, el paralelismo de datos y los tipos inmutables (más adelante en este capítulo), puede reducir la necesidad de bloqueo.



Aquí hay otra forma de percibir el problema: cuando llamas a otro código mientras mantienes un bloqueo, la encapsulación de ese bloqueo se filtra sutilmente. Esto no es un error

en el CLR; es una limitación fundamental del bloqueo en general. Los problemas de bloqueo se están abordando en varios proyectos de investigación, incluida la Memoria Transaccional de Software.

Otro escenario de bloqueo mutuo surge al llamar a Dispatcher.Invoke (en una aplicación WPF) o Control.Invoke (en una aplicación Windows Forms) mientras se posee un bloqueo. Si la interfaz de usuario está ejecutando otro método que está esperando el mismo bloqueo, se producirá un bloqueo mutuo justo ahí. A menudo se puede solucionar esto simplemente llamando a BeginInvoke en lugar de Invoke (o confiando en funciones asíncronas que hacen esto implícitamente cuando hay un contexto de sincronización). Alternativamente, puedes liberar tu bloqueo antes de llamar a Invoke, aunque esto no funcionará si tu llamador ha tomado el bloqueo.

Rendimiento

El bloqueo es rápido: puedes esperar adquirir y liberar un bloqueo en menos de 20 nanosegundos en una computadora de la era 2020 si el bloqueo no está siendo disputado. Si está siendo disputado, el cambio de contexto consecuente acerca la sobrecarga a la región de microsegundos, aunque puede pasar más tiempo antes de que el subproceso se reprograma realmente.

Mutex

Un Mutex es similar a un bloqueo en C#, pero puede funcionar en varios procesos. En otras palabras, un Mutex puede ser a nivel de computadora y también a nivel de aplicación. Adquirir y liberar un Mutex sin disputas lleva alrededor de medio microsegundo, más de 20 veces más lento que un bloqueo.

Con la clase Mutex, llamas al método WaitOne para bloquear y ReleaseMutex para desbloquear. Al igual que con la instrucción lock, un Mutex solo puede ser liberado desde el mismo subproceso que lo obtuvo.


Si olvidas llamar a ReleaseMutex y simplemente llamas a Close o Dispose, se lanzará una excepción AbandonedMutexException cuando alguien más esté esperando ese mutex.

Un uso común para un Mutex entre procesos es garantizar que solo se pueda ejecutar una instancia de un programa a la vez. Así es como se hace:

```
// Naming a Mutex makes it available computer-wide. Use a name that's
// unique to your company and application (e.g., include your URL).

using var mutex = new Mutex (true, @"Global\oreilly.com OneAtATimeDemo");
// Wait a few seconds if contended, in case another instance
// of the program is still in the process of shutting down.
```

```
if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false))
{
  Console.WriteLine ("Another instance of the app is running. Bye!");
  return;
}
try { RunProgram(); }
finally { mutex.ReleaseMutex (); }

void RunProgram()
{
  Console.WriteLine ("Running. Press Enter to exit");
  Console.ReadLine();
}
```



Si estás ejecutando bajo Terminal Services o en consolas Unix separadas, un Mutex a nivel de computadora normalmente solo es visible para las aplicaciones en la misma sesión. Para hacerlo visible en todas las sesiones del servidor de terminales, agrega el prefijo Global\ a su nombre, como se muestra en el ejemplo.

Bloqueo y seguridad de hilos

Un programa o método es seguro para subprocesos si puede funcionar correctamente en cualquier escenario de subprocesamiento múltiple. La seguridad de los subprocesos se logra principalmente mediante el bloqueo y al reducir las posibilidades de interacción entre subprocesos.

Los tipos de propósito general rara vez son seguros para subprocesos en su totalidad, por las siguientes razones:

- La carga de desarrollo en seguridad de subprocesos completa puede ser significativa, especialmente si un tipo tiene muchos campos (cada campo es un potencial para interacción en un contexto multihilo arbitrario).
- La seguridad de subprocesos puede conllevar un costo de rendimiento (pagadero, en parte, tanto si el tipo se utiliza realmente por múltiples subprocesos como si no).
- Un tipo seguro para subprocesos no necesariamente hace que el programa que lo utiliza sea seguro para subprocesos, y a menudo el trabajo involucrado en este último hace que el primero sea redundante.

Por lo tanto, la seguridad de subprocesos generalmente se implementa solo donde es necesario para manejar un escenario de subprocesamiento múltiple específico.

Sin embargo, hay algunas formas de "engañar" y hacer que clases grandes y complejas se ejecuten de manera segura en un entorno multihilo. Una forma es sacrificar la granularidad envolviendo secciones grandes de código, incluso el acceso a un objeto completo, dentro de un único bloqueo exclusivo, lo que impone un acceso serializado a un nivel alto. Esta táctica es, de hecho, esencial si deseas utilizar código de terceros no seguro para subprocesos (o la mayoría de los tipos .NET, en realidad) en un contexto multihilo. El truco consiste simplemente en utilizar el mismo bloqueo exclusivo para proteger el acceso a todas las propiedades, métodos y campos

del objeto no seguro para subprocesos. La solución funciona bien si los métodos del objeto se ejecutan rápidamente (de lo contrario, habrá mucho bloqueo).



Aparte de los tipos primitivos, pocos tipos de .NET, cuando se instancian, son seguros para subprocesos para algo más que un acceso concurrente de solo lectura. La responsabilidad recae en el desarrollador de imponer la seguridad de subprocesos, típicamente con bloqueos exclusivos. (Las colecciones en System.Collections.Concurrent que se cubren en el Capítulo 22 son una excepción).

Otra forma de "engañar" es minimizar la interacción entre subprocesos al minimizar los datos compartidos. Este enfoque es excelente y se utiliza implícitamente en los servidores de aplicaciones y páginas web "sin estado" ("stateless"). Debido a que pueden llegar varias solicitudes de clientes simultáneamente, los métodos del servidor a los que llaman deben ser seguros para subprocesos. Un diseño sin estado (popular por razones de escalabilidad) limita intrínsecamente la posibilidad de interacción porque las clases no guardan datos entre solicitudes. La interacción entre subprocesos se limita entonces solo a los campos estáticos que puedes elegir crear, con el propósito de almacenar en caché datos comúnmente utilizados en la memoria y proporcionar servicios de infraestructura como autenticación y auditoría.

Otra solución (en aplicaciones de cliente enriquecidas) es ejecutar el código que accede al estado compartido en el subproceso de la interfaz de usuario. Como vimos en el Capítulo 14, las funciones asíncronas hacen esto fácilmente.

Seguridad de subprocesos y tipos de .NET

Puedes usar bloqueos para convertir código no seguro para subprocesos en código seguro para subprocesos. Una buena aplicación de esto es .NET: casi todos sus tipos no primitivos no son seguros para subprocesos (para algo más que un acceso de solo lectura) cuando se instancian, pero aún puedes utilizarlos en código multihilo si todo el acceso a un objeto dado está protegido mediante un bloqueo.

Aquí tienes un ejemplo en el que dos subprocesos agregan simultáneamente un elemento a la misma colección List y luego enumeran la lista:

```
class ThreadSafe
{
  static List <string> _list = new List <string>();

  static void Main()
  {
    new Thread (AddItem).Start();
    new Thread (AddItem).Start();
  }

  static void AddItem()
  {
    lock (_list) _list.Add ("Item " + _list.Count);

    string[] items;
    lock (_list) items = _list.ToArray();
    foreach (string s in items) Console.WriteLine (s);
  }
}
```

En este caso, estamos aplicando un bloqueo en el objeto `_list` en sí mismo. Si tuviéramos dos listas interrelacionadas, necesitaríamos elegir un objeto común sobre el cual aplicar el bloqueo (podríamos nominar una de las listas o, mejor aún, usar un campo independiente).

Enumerar colecciones de .NET también es inseguro en términos de hilos, en el sentido de que se lanzará una excepción si la lista se modifica durante la enumeración. En lugar de aplicar el bloqueo durante toda la enumeración, en este ejemplo, primero copiamos los elementos a un array. Esto evita mantener el bloqueo por demasiado tiempo si lo que estamos haciendo durante la enumeración puede llevar mucho tiempo. (Otra solución es usar un bloqueo de lectura/escritura; ver "Bloqueos de Lectura/Escritura" en la página 882).

Bloqueo alrededor de objetos seguros para subprocesos

Aplicar bloqueos alrededor de objetos seguros para subprocesos también es necesario en algunos casos. Para ilustrar esto, imaginemos que la clase `List` de .NET fuera realmente segura para subprocesos y queremos agregar un elemento a una lista:

```
if (!_list.Contains (newItem)) _list.Add (newItem);
```

¡Independientemente de si la lista es segura para subprocesos o no, esta declaración ciertamente no lo es! Toda la instrucción `if` debería estar envuelta en un bloqueo para evitar la posibilidad de que se interrumpa entre la verificación de pertenencia y la adición del nuevo elemento. Este mismo bloqueo debería usarse en cualquier lugar donde modifiquemos esa lista. Por ejemplo, la siguiente instrucción también debería estar envuelta en el mismo bloqueo para asegurarnos de que no interrumpa la declaración anterior:

```
_list.Clear();
```

En otras palabras, necesitaríamos aplicar el bloqueo de la misma manera que con nuestras clases de colección no seguras para subprocesos (lo que haría redundante la supuesta seguridad para subprocesos de la clase `List`).



Aplicar bloqueos alrededor del acceso a una colección puede causar bloqueos excesivos en entornos altamente concurrentes. Para solucionar esto, .NET proporciona una cola (queue), una pila (stack) y un diccionario (dictionary) seguros para subprocesos, los cuales se discuten en el Capítulo 22.

Miembros estáticos

Envolver el acceso a un objeto en un bloqueo personalizado solo funciona si todos los subprocesos concurrentes son conscientes del bloqueo y lo utilizan. Esto puede no ser el caso si el objeto tiene un alcance amplio. El peor caso se da con los miembros estáticos en un tipo público. Por ejemplo, imaginemos que la propiedad estática en la estructura `DateTime`, `DateTime.Now`, no es segura para subprocesos y que dos llamadas concurrentes podrían resultar en una salida incorrecta o una excepción. La única forma de solucionar esto con bloqueos externos sería bloquear el tipo en sí mismo (`lock(typeof(DateTime))`) antes de llamar

a `DateTime.Now`. Esto solo funcionaría si todos los programadores acuerdan hacerlo (lo cual es poco probable). Además, bloquear un tipo crea problemas propios.

Por esta razón, los miembros estáticos en la estructura DateTime han sido programados cuidadosamente para ser seguros para subprocesos. Este es un patrón común en todo .NET: los miembros estáticos son seguros para subprocesos; los miembros de instancia no lo son. Seguir este patrón también tiene sentido al escribir tipos para consumo público, para no crear dilemas imposibles de seguridad de subprocesos. En otras palabras, al hacer que los métodos estáticos sean seguros para subprocesos, estás programando de manera que no excluyas la seguridad de subprocesos para los consumidores de ese tipo.


La seguridad de subprocesos en métodos estáticos es algo que debes codificar explícitamente: no ocurre automáticamente por ser el método estático.

La seguridad de solo lectura para subprocesos

Hacer que los tipos sean seguros para subprocesos para acceso concurrente de solo lectura (si es posible) es ventajoso porque significa que los consumidores pueden evitar un bloqueo excesivo. Muchos tipos en .NET siguen este principio: las colecciones, por ejemplo, son seguras para subprocesos en lectura concurrente.

Seguir este principio es simple: si documentas un tipo como seguro para subprocesos para acceso de solo lectura concurrente, no escribas en campos dentro de métodos que un consumidor esperaría que sean de solo lectura (o bloquea al hacerlo). Por ejemplo, al implementar un método ToArray() en una colección, puedes comenzar compactando la estructura interna de la colección. Sin embargo, esto haría que no sea seguro para subprocesos para los consumidores que esperan que sea de solo lectura.

La seguridad de solo lectura para subprocesos es una de las razones por las que los enumeradores son separados de los "enumerables": dos subprocesos pueden enumerar simultáneamente sobre una colección porque cada uno obtiene un objeto enumerador separado.


En ausencia de documentación, es prudente ser cauteloso al asumir si un método es de naturaleza de solo lectura. Un buen ejemplo es la clase Random: cuando llamas a Random.Next(), su implementación interna requiere que actualice valores de semilla privados. Por lo tanto, debes bloquear al usar la clase Random o mantener una instancia separada por subproceso.

Seguridad de subprocesos en servidores de aplicaciones (Thread Safety in Application Servers)

Los servidores de aplicaciones deben ser multinúcleo para manejar solicitudes de clientes simultáneas. Las aplicaciones ASP.NET Core y Web API son implícitamente multinúcleo. Esto significa que al escribir código en el lado del servidor, debes considerar la seguridad de subprocesos si existe alguna posibilidad de interacción entre los subprocesos que procesan las solicitudes de los clientes. Afortunadamente, tal posibilidad es rara; una clase de servidor típica es o bien sin estado (sin campos) o tiene un modelo de activación que crea una instancia de objeto separada para cada cliente o cada solicitud. La interacción generalmente surge solo a través de campos estáticos, a veces utilizados para almacenar en caché en memoria partes de una base de datos para mejorar el rendimiento.

Por ejemplo, supongamos que tienes un método RetrieveUser que consulta una base de datos:

```
// User is a custom class with fields for user data
internal User RetrieveUser (int id) { ... }
```

Si este método se llama con frecuencia, puedes mejorar el rendimiento almacenando en caché los resultados en un Dictionary estático. Aquí tienes una solución conceptualmente simple que tiene en cuenta la seguridad de subprocesos:

```
static class UserCache
{
  static Dictionary <int, User> _users = new Dictionary <int, User>();

  internal static User GetUser (int id)
  {
    User u = null;

    lock (_users)
      if (_users.TryGetValue (id, out u))
        return u;

    u = RetrieveUser (id);           // Method to retrieve from database;
    lock (_users) _users [id] = u;
    return u;
  }
}
```

Como mínimo, debemos bloquear la lectura y actualización del diccionario para garantizar la seguridad de subprocesos. En este ejemplo, elegimos un compromiso práctico entre simplicidad y rendimiento en el bloqueo. Nuestro diseño crea una pequeña posibilidad de ineficiencia: si dos subprocesos llaman simultáneamente a este método con el mismo id no recuperado previamente, el método RetrieveUser se llamaría dos veces y el diccionario se actualizaría innecesariamente. Bloquear una vez en todo el método evitaría esto, pero crearía una mayor ineficiencia: todo el caché se bloquearía durante la duración de la llamada a RetrieveUser, durante la cual otros subprocesos estarían bloqueados al intentar recuperar cualquier usuario.

Para una solución ideal, debemos utilizar la estrategia que describimos en "Completando de forma síncrona" en la página 655. En lugar de almacenar en caché un objeto User, almacenamos un Task<User>, que el llamador luego espera:

```
static class UserCache
{
  static Dictionary <int, Task<User>> _userTasks =
     new Dictionary <int, Task<User>>();

  internal static Task<User> GetUserAsync (int id)
  {
    lock (_userTasks)
      if (_userTasks.TryGetValue (id, out var userTask))
        return userTask;
      else
        return _userTasks [id] = Task.Run (() => RetrieveUser (id));
  }
}
```

Observa que ahora tenemos un solo bloqueo que cubre toda la lógica del método. Podemos hacer esto sin afectar la concurrencia porque todo lo que estamos haciendo dentro del bloqueo es acceder al diccionario y (potencialmente) iniciar una operación asíncrona (llamando a Task.Run). Si dos subprocesos llaman a este método al mismo tiempo con el mismo ID, ambos esperarán la misma tarea, que es exactamente el resultado que queremos.

Objetos inmutables

Un objeto inmutable es aquel cuyo estado no puede ser alterado, ni externa ni internamente. Los campos de un objeto inmutable generalmente se declaran como de solo lectura y se inicializan por completo durante la construcción.

La inmutabilidad es una característica de la programación funcional, donde en lugar de mutar un objeto, se crea un nuevo objeto con propiedades diferentes. LINQ sigue este paradigma. La inmutabilidad también es valiosa en la programación multihilo, ya que evita el problema de tener un estado compartido modificable al eliminar (o minimizar) lo modificable.

Un patrón es utilizar objetos inmutables para encapsular un grupo de campos relacionados y minimizar la duración de los bloqueos. Tomemos un ejemplo muy sencillo, supongamos que tenemos dos campos de la siguiente manera:

```
int _percentComplete;
string _statusMessage;
```

Ahora supongamos que queremos leer y escribir estos campos de forma atómica. En lugar de bloquear en torno a estos campos, podríamos definir la siguiente clase inmutable:

```
class ProgressStatus    // Represents progress of some activity
{
  public readonly int PercentComplete;
  public readonly string StatusMessage;

  // This class might have many more fields...

  public ProgressStatus (int percentComplete, string statusMessage)
  {
    PercentComplete = percentComplete;
    StatusMessage = statusMessage;
  }
}
```

Luego podríamos definir un solo campo de ese tipo, junto con un objeto de bloqueo:

```
readonly object _statusLocker = new object();
ProgressStatus _status;
```

Ahora podemos leer y escribir valores de ese tipo sin mantener un bloqueo durante más de una asignación:

```
var status = new ProgressStatus (50, "Working on it");
// Imagine we were assigning many more fields...
// ...
lock (_statusLocker) _status = status;    // Very brief lock
```

Para leer el objeto, primero obtenemos una copia de la referencia del objeto (dentro de un bloqueo). Luego, podemos leer sus valores sin necesidad de mantener el bloqueo:

```
ProgressStatus status;
lock (_statusLocker) status = _status;   // Again, a brief lock
int pc = status.PercentComplete;
string msg = status.StatusMessage;
...
```

Bloqueo no exclusivo

Las construcciones de bloqueo no exclusivo sirven para limitar la concurrencia. En esta sección, cubrimos los semáforos y los bloqueos de lectura/escritura, y también ilustramos cómo la clase SemaphoreSlim puede limitar la concurrencia con operaciones asíncronas.

Semáforo

Un semáforo es como una discoteca: tiene una capacidad determinada, aplicada por un portero. Cuando la discoteca está llena, no se permite la entrada de más personas y se forma una cola en el exterior. Luego, por cada persona que sale, una persona entra. El constructor requiere un mínimo de dos argumentos: el número de lugares actualmente disponibles en la discoteca y la capacidad total de la discoteca.

Un semáforo con una capacidad de uno es similar a un Mutex o un bloqueo (lock), excepto que el semáforo no tiene un "propietario" específico: es independiente del subproceso. Cualquier subproceso puede llamar a Release en un semáforo, mientras que con Mutex y lock, solo el subproceso que obtuvo el bloqueo puede liberarlo.



Existen dos versiones funcionalmente similares de esta clase: Semaphore y SemaphoreSlim. Este último ha sido optimizado para cumplir con las demandas de baja latencia de la programación paralela. También es útil en la programación multihilo tradicional porque te permite especificar un token de cancelación al esperar (ver "Cancelación" en la página 659) y expone un método WaitAsync para la programación asíncrona. Sin embargo, no se puede utilizar para la señalización entre procesos.

El método WaitOne de Semaphore demora aproximadamente un microsegundo en llamarse y en realizarse el Release; SemaphoreSlim demora aproximadamente una décima parte de eso.

Los semáforos pueden ser útiles para limitar la concurrencia, evitando que demasiados subprocesos ejecuten un fragmento de código en particular al mismo tiempo. En el siguiente ejemplo, cinco subprocesos intentan ingresar a una discoteca que solo permite la entrada de tres subprocesos a la vez:

```
class TheClub       // No door lists!
{
  static SemaphoreSlim _sem = new SemaphoreSlim (3);    // Capacity of 3

  static void Main()
  {
    for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);
  }

  static void Enter (object id)
  {


    Console.WriteLine (id + " wants to enter");
    _sem.Wait();
    Console.WriteLine (id + " is in!");           // Only three threads
    Thread.Sleep (1000 * (int) id);               // can be here at
    Console.WriteLine (id + " is leaving");       // a time.
    _sem.Release();
  }
}
```

```
1 wants to enter
1 is in!
2 wants to enter
2 is in!
3 wants to enter
3 is in!
4 wants to enter
5 wants to enter
1 is leaving
4 is in!
2 is leaving
5 is in!
```

Un Semaphore, si tiene un nombre, puede abarcar procesos de la misma manera que un Mutex (los Semaphore con nombre solo están disponibles en Windows, mientras que los Mutex con nombre también funcionan en plataformas Unix).

Semáforos y bloqueos asíncronos

Es ilegal realizar un bloqueo a través de una instrucción await:

```
lock (_locker)
{
  await Task.Delay (1000);    // Compilation error
  ...
}
```

Hacer esto no tendría sentido, porque los bloqueos son mantenidos por un subproceso, que normalmente cambia al regresar de un await. Además, el bloqueo bloquea la ejecución y bloquearse durante un período potencialmente largo de tiempo es exactamente lo que no se intenta lograr con funciones asíncronas.

Sin embargo, a veces sigue siendo deseable que las operaciones asíncronas se ejecuten secuencialmente o limitar la paralelismo de tal manera que no se ejecuten más de n operaciones al mismo tiempo. Por ejemplo, consideremos un navegador web: necesita realizar descargas asíncronas en paralelo, pero puede querer imponer un límite máximo de 10 descargas al mismo tiempo. Podemos lograr esto utilizando un SemaphoreSlim:

```
SemaphoreSlim _semaphore = new SemaphoreSlim (10);

async Task<byte[]> DownloadWithSemaphoreAsync (string uri)
{
    await _semaphore.WaitAsync();
    try { return await new WebClient().DownloadDataTaskAsync (uri); }


    finally { _semaphore.Release(); }
}
```

Reducir el `initialCount` del semáforo a 1 reduce el paralelismo máximo a 1, convirtiéndolo en un bloqueo asíncrono.

Escribir un método de extensión `EnterAsync`

El siguiente método de extensión simplifica el uso asíncrono de `SemaphoreSlim` utilizando la clase `Disposable` que escribimos en "Anonymous Disposal" en la página 561:

```
public static async Task<IDisposable> EnterAsync (this SemaphoreSlim ss)
{
  await ss.WaitAsync().ConfigureAwait (false);
  return Disposable.Create (() => ss.Release());
}
```

Con este método, podemos reescribir nuestro método `DownloadWithSemaphoreAsync` de la siguiente manera:

```
async Task<byte[]> DownloadWithSemaphoreAsync (string uri)
{
  using (await _semaphore.EnterAsync())
    return await new WebClient().DownloadDataTaskAsync (uri);
}
```

Parallel.ForEachAsync

A partir de .NET 6, otra forma de limitar la concurrencia asíncrona es utilizar el método `Parallel.ForEachAsync`. Supongamos que `uris` es un array de URIs que deseamos descargar. Aquí se muestra cómo descargarlos en paralelo, limitando la concurrencia a un máximo de 10 descargas paralelas:

```
await Parallel.ForEachAsync (uris,
  new ParallelOptions { MaxDegreeOfParallelism = 10 },
  async (uri, cancelToken) =>
   {
    var download = await new HttpClient().GetByteArrayAsync (uri);
    Console.WriteLine ($"Downloaded {download.Length} bytes");
  });
```

Los demás métodos en la clase `Parallel` están destinados a escenarios de programación paralela (relacionados con el uso intensivo de la CPU), que se describen en el Capítulo 22.

Bloqueos de lectura/escritura

Con frecuencia, las instancias de un tipo son seguras para operaciones de lectura concurrentes, pero no para actualizaciones concurrentes (ni para una lectura y actualización concurrente). Esto también puede ser cierto con recursos como un archivo. Aunque proteger instancias de este tipo con un bloqueo exclusivo para todos los modos de acceso generalmente funciona, puede restringir de manera irrazonable la concurrencia si hay muchos lectores y solo actualizaciones ocasionales. Un ejemplo de esto puede ocurrir en un servidor de aplicaciones empresariales, donde los datos comúnmente utilizados se almacenan en caché para una recuperación rápida en campos estáticos.

La clase ReaderWriterLockSlim está diseñada para proporcionar bloqueo de máxima disponibilidad en este escenario.



ReaderWriterLockSlim es un reemplazo para la antigua clase "grande" ReaderWriterLock. Esta última es similar en funcionalidad, pero es varias veces más lenta y tiene un defecto inherente en su mecanismo para manejar actualizaciones de bloqueo.

En comparación con un bloqueo ordinario (Monitor.Enter/Exit), ReaderWriterLockSlim sigue siendo el doble de lento. Sin embargo, el compromiso es menor conflicto (cuando hay mucha lectura y escritura mínima).

Con ambas clases, hay dos tipos básicos de bloqueo: un bloqueo de lectura y un bloqueo de escritura:

- Un bloqueo de escritura es exclusivo de forma universal.
- Un bloqueo de lectura es compatible con otros bloqueos de lectura.

Entonces, un hilo que tiene un bloqueo de escritura bloquea a todos los demás hilos que intentan obtener un bloqueo de lectura o escritura (y viceversa). Pero si ningún hilo tiene un bloqueo de escritura, cualquier número de hilos puede obtener simultáneamente un bloqueo de lectura.

ReaderWriterLockSlim define los siguientes métodos para obtener y liberar bloqueos de lectura/escritura:

```
public void EnterReadLock();
public void ExitReadLock();
public void EnterWriteLock();
public void ExitWriteLock();
```

Además, hay versiones "Try" de todos los métodos EnterXXX que aceptan argumentos de tiempo de espera al estilo de Monitor.TryEnter (los tiempos de espera pueden ocurrir fácilmente si el recurso está altamente disputado). ReaderWriterLock proporciona métodos similares llamados AcquireXXX y ReleaseXXX. Estos lanzan una ApplicationException si se produce un tiempo de espera, en lugar de devolver falso.

El siguiente programa demuestra el uso de ReaderWriterLockSlim. Tres hilos enumeran continuamente una lista, mientras que otros dos hilos agregan un número aleatorio a la lista cada 100 ms. Un bloqueo de lectura protege a los lectores de la lista, y un bloqueo de escritura protege a los escritores de la lista:

```
class SlimDemo
{
  static ReaderWriterLockSlim _rw = new ReaderWriterLockSlim();
  static List<int> _items = new List<int>();
  static Random _rand = new Random();

  static void Main()
  {
    new Thread (Read).Start();
    new Thread (Read).Start();
    new Thread (Read).Start();

    new Thread (Write).Start ("A");
    new Thread (Write).Start ("B");
  }

  static void Read()
  {
    while (true)
    {
      _rw.EnterReadLock();
      foreach (int i in _items) Thread.Sleep (10);
      _rw.ExitReadLock();
    }
  }

  static void Write (object threadID)
  {
    while (true)
    {
      int newNumber = GetRandNum (100);
      _rw.EnterWriteLock();
      _items.Add (newNumber);
      _rw.ExitWriteLock();
      Console.WriteLine ("Thread " + threadID + " added " + newNumber);
      Thread.Sleep (100);
    }
  }

  static int GetRandNum (int max) { lock (_rand) return _rand.Next(max); }
}
```



En código de producción, normalmente se añadirían bloques try/finally para asegurarse de que los bloqueos se liberaran si se producía una excepción.

Aquí está el resultado:

```
Thread B added 61
Thread A added 83
Thread B added 55
Thread A added 33
...
```

ReaderWriterLockSlim permite más actividad de lectura concurrente que un simple bloqueo. Podemos ilustrar esto insertando la siguiente línea en el método Write, al comienzo del bucle while:

```
Console.WriteLine (_rw.CurrentReadCount + " concurrent readers");
```

Esto casi siempre imprime "3 lectores concurrentes" (los métodos Read pasan la mayor parte de su tiempo dentro de los bucles foreach). Además de CurrentReadCount, ReaderWriterLockSlim proporciona las siguientes propiedades para supervisar los bloqueos:

```
public bool IsReadLockHeld            { get; }
public bool IsUpgradeableReadLockHeld { get; }
public bool IsWriteLockHeld           { get; }

public int  WaitingReadCount          { get; }
public int  WaitingUpgradeCount       { get; }
public int  WaitingWriteCount         { get; }

public int  RecursiveReadCount        { get; }
public int  RecursiveUpgradeCount     { get; }
public int  RecursiveWriteCount       { get; }
```

Bloqueos actualizables

En ocasiones, resulta útil cambiar un bloqueo de lectura por un bloqueo de escritura en una única operación atómica. Por ejemplo, supongamos que deseamos añadir un elemento a una lista solo si el elemento no está presente. Idealmente, queremos minimizar el tiempo en que mantenemos el bloqueo de escritura exclusivo, por lo que podríamos seguir los siguientes pasos:

1. Obtener un bloqueo de lectura.

2. Comprobar si el elemento ya está presente en la lista; si es así, liberar el bloqueo y retornar.

3. Liberar el bloqueo de lectura.

4. Obtener un bloqueo de escritura.

5. Añadir el elemento.

El problema es que otro hilo podría aprovechar la oportunidad y modificar la lista (por ejemplo, añadir el mismo elemento) entre los pasos 3 y 4. ReaderWriterLockSlim aborda esto mediante un tercer tipo de bloqueo llamado bloqueo actualizable. Un bloqueo actualizable es similar a un bloqueo de lectura, excepto que puede promocionarse posteriormente a un bloqueo de escritura en una operación atómica. Así es cómo se utiliza:

1. Llamar a EnterUpgradeableReadLock.

2. Realizar actividades basadas en la lectura (por ejemplo, comprobar si el elemento ya está presente en la lista).

3. Llamar a EnterWriteLock (esto convierte el bloqueo actualizable en un bloqueo de escritura).

4. Realizar actividades basadas en la escritura (por ejemplo, añadir el elemento a la lista).

5. Llamar a ExitWriteLock (esto convierte el bloqueo de escritura de nuevo en un bloqueo actualizable).

6. Realizar cualquier otra actividad basada en la lectura.

7. Llamar a ExitUpgradeableReadLock.

Desde la perspectiva del llamador, es similar a un bloqueo anidado o recursivo. Sin embargo, funcionalmente, en el paso 3, ReaderWriterLockSlim libera el bloqueo de lectura y obtiene un nuevo bloqueo de escritura de manera atómica.

Hay otra diferencia importante entre los bloqueos actualizables y los bloqueos de lectura. Aunque un bloqueo actualizable puede coexistir con cualquier número de bloqueos de lectura, solo se puede obtener un bloqueo actualizable a la vez. Esto evita bloqueos de conversión en serie y posibles bloqueos muertos, al igual que los bloqueos de actualización en SQL Server:

SQL Server	ReaderWriterLockSlim
Share lock	Read lock
Exclusive lock	Write lock
Update lock	Upgradeable lock

Podemos demostrar un bloqueo actualizable cambiando el método Write en el ejemplo anterior para que añada un número a la lista solo si aún no está presente:

```
while (true)
{
  int newNumber = GetRandNum (100);
  _rw.EnterUpgradeableReadLock();
  if (!_items.Contains (newNumber))
  {
    _rw.EnterWriteLock();
    _items.Add (newNumber);
    _rw.ExitWriteLock();
    Console.WriteLine ("Thread " + threadID + " added " + newNumber);
  }
  _rw.ExitUpgradeableReadLock();
  Thread.Sleep (100);
}
```



ReaderWriterLock también puede realizar conversiones de bloqueo, pero de manera poco confiable debido a que no admite el concepto de bloqueos actualizables. Por esta razón, los diseñadores de ReaderWriterLockSlim tuvieron que empezar de nuevo con una nueva clase.

Recursión de bloqueo

Normalmente, los bloqueos anidados o recursivos están prohibidos con ReaderWriterLockSlim. Por lo tanto, el siguiente código genera una excepción:

```
var rw = new ReaderWriterLockSlim();
rw.EnterReadLock();
rw.EnterReadLock();
rw.ExitReadLock();
rw.ExitReadLock();
```

Sin embargo, se ejecuta sin errores si se construye ReaderWriterLockSlim de la siguiente manera:

```
var rw = new ReaderWriterLockSlim (LockRecursionPolicy.SupportsRecursion);
```

Esto asegura que el bloqueo recursivo solo puede ocurrir si lo planificas. El bloqueo recursivo puede crear una complejidad no deseada porque es posible adquirir más de un tipo de bloqueo:

```
rw.EnterWriteLock();
rw.EnterReadLock();
Console.WriteLine (rw.IsReadLockHeld);      // True
Console.WriteLine (rw.IsWriteLockHeld);     // True
rw.ExitReadLock();
rw.ExitWriteLock();
```

La regla básica es que después de adquirir un bloqueo, los bloqueos recursivos subsiguientes pueden ser inferiores, pero no superiores, en la siguiente escala:

Read Lock→Upgradeable Lock→Write Lock

Sin embargo, siempre es válido solicitar la promoción de un bloqueo actualizable a un bloqueo de escritura.

Señalización con Event Wait Handles

El tipo más simple de constructos de señalización se llama "event wait handles" (no relacionado con los eventos de C#). Los event wait handles vienen en tres variantes: AutoResetEvent, ManualResetEvent(Slim) y CountdownEvent. Los dos primeros se basan en la clase común EventWaitHandle, de la cual derivan toda su funcionalidad.

AutoResetEvent

Un AutoResetEvent es como un torniquete de boletos: insertar un boleto permite que exactamente una persona pase. El "auto" en el nombre de la clase se refiere al hecho de que un torniquete abierto se cierra automáticamente o "se reinicia" después de que una persona haya pasado. Un hilo espera, o se bloquea, en el torniquete llamando a WaitOne (esperar en este torniquete "uno" hasta que se abra) y se inserta un boleto llamando al método Set. Si varios hilos llaman a WaitOne, se forma una cola detrás del torniquete. Un boleto puede provenir de cualquier hilo; en otras palabras, cualquier hilo desbloqueado con acceso al objeto AutoResetEvent puede llamar a Set en él para liberar a un hilo bloqueado.

Puedes crear un AutoResetEvent de dos formas. La primera es a través de su constructor:

```
var auto = new AutoResetEvent (false);
```

(Pasar true en el constructor es equivalente a llamar inmediatamente a Set en él).

La segunda forma de crear un AutoResetEvent es la siguiente:

```
var auto = new EventWaitHandle (false, EventResetMode.AutoReset);
```

En el siguiente ejemplo, se inicia un hilo cuyo trabajo consiste en esperar hasta que sea señalado por otro hilo (ver Figura 21-1):

```
class BasicWaitHandle
{
  static EventWaitHandle _waitHandle = new AutoResetEvent (false);

  static void Main()
  {
    new Thread (Waiter).Start();
    Thread.Sleep (1000);                  // Pause for a second...
    _waitHandle.Set();                    // Wake up the Waiter.
  }

  static void Waiter()
  {
    Console.WriteLine ("Waiting...");
    _waitHandle.WaitOne();                // Wait for notification
    Console.WriteLine ("Notified");
  }
}

// Output:
Waiting... (pause) Notified.
```

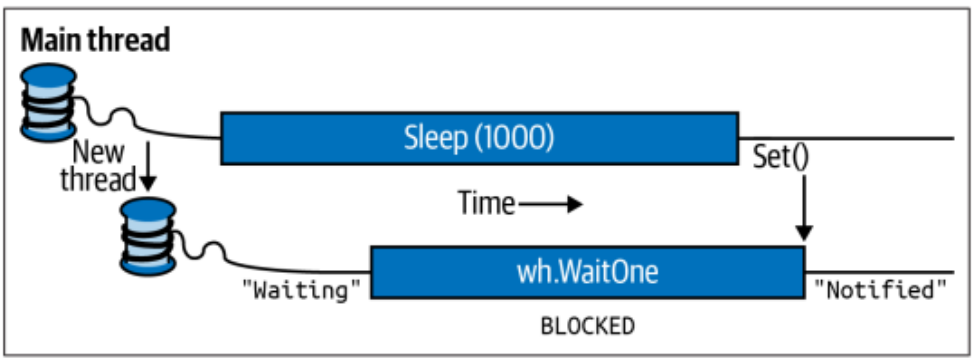


Figure 21-1. Signaling with an EventWaitHandle

Si se llama a `Set` cuando no hay ningún hilo esperando, el evento permanecerá abierto hasta que algún hilo llame a `WaitOne`. Este comportamiento ayuda a evitar una carrera entre un hilo que se dirige hacia el torniquete y un hilo que inserta un boleto ("¡Ups, inserté el boleto un microsegundo demasiado pronto; ahora tendrás que esperar indefinidamente!"). Sin embargo, llamar repetidamente a `Set` en un torniquete en el que nadie está esperando no permite que todo un grupo pase cuando llegan: solo se permite el paso de la siguiente persona y los boletos adicionales se "desperdician".

Eliminando los Wait Handles (Disposing Wait Handles)

Después de haber terminado con un wait handle, puedes llamar a su método Close para liberar el recurso del sistema operativo. Alternativamente, puedes simplemente dejar que todas las referencias al wait handle se pierdan y permitir que el recolector de basura haga el trabajo por ti en algún momento posterior (los wait handles implementan el patrón de disposición en el que el finalizador llama a Close). Este es uno de los pocos escenarios en los que confiar en esta opción de respaldo es (discutiblemente) aceptable, porque los wait handles tienen una carga ligera para el sistema operativo.

Los wait handles se liberan automáticamente cuando un proceso se cierra.

Llamando a Reset en un AutoResetEvent cierra el torno (si está abierto) sin esperar ni bloquear.

WaitOne acepta un parámetro opcional de tiempo de espera, devolviendo false si la espera termina debido a un tiempo de espera en lugar de obtener la señal.



Llamar a WaitOne con un tiempo de espera de 0 permite probar si un wait handle está "abierto", sin bloquear al llamador. Sin embargo, hay que tener en cuenta que hacer esto restablece el AutoResetEvent si está abierto.

Señalización bidireccional

Supongamos que queremos que el hilo principal señale al hilo de trabajo tres veces seguidas. Si el hilo principal simplemente llama a Set en un wait handle varias veces en rápida sucesión, la segunda o tercera señal puede perderse porque el hilo de trabajo puede tardar en procesar cada señal.

La solución es que el hilo principal espere hasta que el hilo de trabajo esté listo antes de señalarlo. Podemos hacer esto utilizando otro AutoResetEvent, de la siguiente manera:

```
class TwoWaySignaling
{
  static EventWaitHandle _ready = new AutoResetEvent (false);
  static EventWaitHandle _go = new AutoResetEvent (false);
  static readonly object _locker = new object();
  static string _message;

  static void Main()
  {
    new Thread (Work).Start();

    _ready.WaitOne();                  // First wait until worker is ready
    lock (_locker) _message = "ooo";
    _go.Set();                         // Tell worker to go

    _ready.WaitOne();
    lock (_locker) _message = "ahhh";  // Give the worker another message
    _go.Set();

    _ready.WaitOne();

      lock (_locker) _message = null;    // Signal the worker to exit
      _go.Set();
    }

    static void Work()
    {
      while (true)
      {
        _ready.Set();                          // Indicate that we're ready
        _go.WaitOne();                         // Wait to be kicked off...
        lock (_locker)
        {
          if (_message == null) return;        // Gracefully exit
          Console.WriteLine (_message);
        }
      }
    }
  }

  // Output:
  ooo
  ahhh
```

Figure 21-2 shows this process.

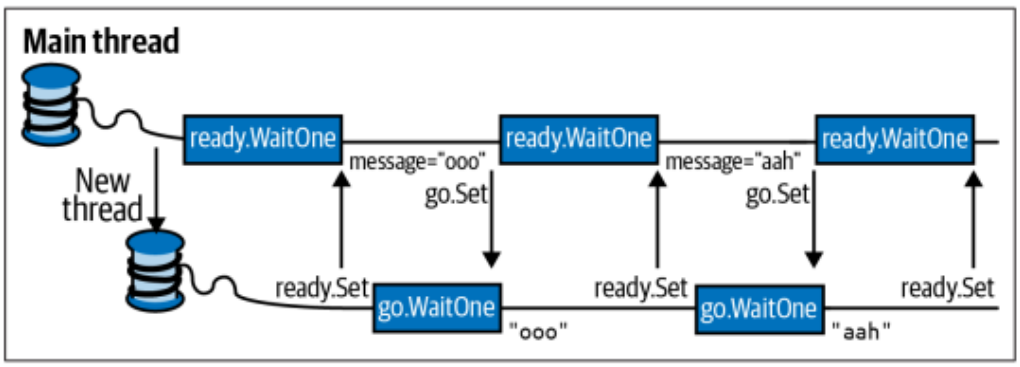


Figure 21-2. Two-way signaling

Aquí, estamos utilizando un mensaje nulo para indicar que el trabajador debería finalizar. ¡Con hilos que se ejecutan indefinidamente, es importante tener una estrategia de salida!

ManualResetEvent

Como describimos en el Capítulo 14, un ManualResetEvent funciona como una simple puerta. Llamar a Set abre la puerta, permitiendo que cualquier número de hilos que llamen a WaitOne sean liberados. Llamar a Reset cierra la puerta. Los hilos que llaman a WaitOne en una puerta cerrada se bloquearán; cuando la puerta se abra nuevamente, serán liberados todos a la vez. Aparte de estas diferencias, un ManualResetEvent funciona como un AutoResetEvent.

Al igual que con AutoResetEvent, puedes construir un ManualResetEvent de dos formas:

```
var manual1 = new ManualResetEvent (false);
var manual2 = new EventWaitHandle (false, EventResetMode.ManualReset);
```



Hay otra versión de ManualResetEvent llamada ManualResetEventSlim. Esta última está optimizada para tiempos de espera cortos, con la capacidad de activar el giro durante un número determinado de iteraciones. También tiene una implementación administrada más eficiente y permite cancelar una espera mediante un CancellationToken.

ManualResetEventSlim no hereda de WaitHandle; sin embargo, expone una propiedad WaitHandle que devuelve un objeto basado en WaitHandle cuando se llama (con el perfil de rendimiento de un wait handle tradicional).

Estructuras de señalización y rendimiento (Signaling Constructs and Performance)

Las operaciones de espera o señalización en un AutoResetEvent o ManualResetEvent toman aproximadamente un microsegundo (suponiendo que no haya bloqueo).

ManualResetEventSlim y CountdownEvent pueden ser hasta 50 veces más rápidos en escenarios de espera corta debido a su no dependencia del sistema operativo y al uso juicioso de constructos de giro. Sin embargo, en la mayoría de los escenarios, el costo adicional de las clases de señalización en sí mismas no crea un cuello de botella, por lo que rara vez es una consideración.

Un ManualResetEvent es útil para permitir que un hilo desbloquee a muchos otros hilos. El escenario inverso está cubierto por CountdownEvent.

CountdownEvent

CountdownEvent permite esperar en más de un hilo. La clase tiene una implementación eficiente completamente administrada. Para usar la clase, instantíala con la cantidad de hilos, o "cuentas", en los que deseas esperar:

```
var countdown = new CountdownEvent (3);  // Initialize with "count" of 3.
```

Llamar a Signal decrementa el "contador"; llamar a Wait bloquea hasta que el contador llega a cero:

```
new Thread (SaySomething).Start ("I am thread 1");
new Thread (SaySomething).Start ("I am thread 2");
new Thread (SaySomething).Start ("I am thread 3");

countdown.Wait();   // Blocks until Signal has been called 3 times
Console.WriteLine ("All threads have finished speaking!");

void SaySomething (object thing)
{
  Thread.Sleep (1000);
  Console.WriteLine (thing);
  countdown.Signal();
}
```



Puede resolver problemas de manera más fácil utilizando los constructos de paralelismo estructurado que describimos en el Capítulo 22 (PLINQ y la clase Parallel).

Puede reincrementar el contador de un CountdownEvent llamando a AddCount. Sin embargo, si ya ha alcanzado cero, esto genera una excepción: no se puede "desactivar" un CountdownEvent llamando a AddCount. Para evitar la posibilidad de que se genere una excepción, en su lugar puede llamar a TryAddCount, que devuelve false si la cuenta regresiva es cero.

Para desactivar un evento de cuenta regresiva, llame a Reset: esto desactiva la estructura y restablece su contador al valor original.

Al igual que ManualResetEventSlim, CountdownEvent expone una propiedad WaitHandle para escenarios en los que alguna otra clase o método espera un objeto basado en WaitHandle.

Creación de un EventWaitHandle entre procesos

El constructor de EventWaitHandle permite crear un EventWaitHandle "nombrado" capaz de funcionar en múltiples procesos. El nombre es simplemente una cadena y puede ser cualquier valor que no entre en conflicto accidentalmente con otros. Si el nombre ya está en uso en el equipo, obtendrá una referencia al mismo EventWaitHandle subyacente; de lo contrario, el sistema operativo creará uno nuevo. Aquí hay un ejemplo:

```
EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.AutoReset,
                                          @"Global\MyCompany.MyApp.SomeName");
```

Si dos aplicaciones ejecutan este código, podrán señalarse entre sí: el wait handle funcionará en todos los subprocesos de ambos procesos.

Los wait handles nombrados solo están disponibles en Windows.

Wait Handles y Continuaciones

En lugar de esperar en un wait handle (y bloquear su hilo), puede adjuntar una "continuación" a él llamando a ThreadPool.RegisterWaitForSingleObject. Este método acepta un delegado que se ejecuta cuando se señaliza un wait handle:

```
var starter = new ManualResetEvent (false);

RegisteredWaitHandle reg = ThreadPool.RegisterWaitForSingleObject
 (starter, Go, "Some Data", -1, true);

Thread.Sleep (5000);
Console.WriteLine ("Signaling worker...");
starter.Set();
Console.ReadLine();
reg.Unregister (starter);    // Clean up when we're done.

void Go (object data, bool timedOut)


{
  Console.WriteLine ("Started - " + data);
  // Perform task...
}

// Output:
(5 second delay)
Signaling worker...
Started - Some Data
```

Cuando se señaliza el wait handle (o transcurre un tiempo de espera), el delegado se ejecuta en un hilo del grupo de hilos. Luego, se supone que debe llamar a Unregister para liberar el controlador no administrado de la devolución de llamada.

Además del wait handle y el delegado, RegisterWaitForSingleObject acepta un objeto "caja negra" que se pasa al método de delegado (similar a ParameterizedThreadStart), así como un tiempo de espera en milisegundos (-1 significa sin tiempo de espera) y un indicador booleano que indica si la solicitud es única o recurrente.



Solo se puede llamar de manera confiable a RegisterWaitForSingleObject una vez por wait handle. Llamar a este método nuevamente en el mismo wait handle provoca un fallo intermitente, en el que un wait handle sin señalizar dispara una devolución de llamada como si estuviera señalizado.

Esta limitación hace que los wait handles (no slim) sean poco adecuados para la programación asíncrona.

WaitAny, WaitAll y SignalAndWait

Además de los métodos Set, WaitOne y Reset, existen métodos estáticos en la clase WaitHandle para resolver problemas de sincronización más complejos. Los métodos WaitAny, WaitAll y SignalAndWait realizan operaciones de señalización y espera en múltiples handles. Los wait handles pueden ser de diferentes tipos (incluyendo Mutex y Semaphore, dado que también derivan de la clase abstracta WaitHandle).

ManualResetEventSlim y CountdownEvent también pueden participar en estos métodos a través de sus propiedades WaitHandle.



WaitAll y SignalAndWait están relacionados de manera extraña con la arquitectura COM heredada: estos métodos requieren que el llamador esté en un apartamento de subprocesos múltiples, que es el modelo menos adecuado para la interoperabilidad. El hilo principal de una aplicación WPF o Windows Forms, por ejemplo, no puede interactuar con el portapapeles en este modo. Discutiremos alternativas en breve.

WaitHandle.WaitAny espera a que se active cualquiera de los wait handles de una matriz; WaitHandle.WaitAll espera a que se activen todos los handles dados, de forma atómica. Esto significa que si espera en dos AutoResetEvents:

- WaitAny nunca terminará "latcheando" ambos eventos.
- WaitAll nunca terminará "latcheando" solo un evento.
- SignalAndWait llama a Set en un WaitHandle y luego llama a WaitOne en otro WaitHandle.

Después de señalizar el primer handle, saltará al principio de la cola para esperar en el segundo handle; esto ayuda a que tenga éxito (aunque la operación no es realmente atómica). Puedes pensar en este método como un "intercambio" de una señal por otra, y usarlo en un par de EventWaitHandles para configurar dos hilos para que se encuentren en el mismo punto en el tiempo. Tanto AutoResetEvent como ManualResetEvent funcionarán. El primer hilo ejecuta lo siguiente:

```
WaitHandle.SignalAndWait (wh1, wh2);
```

El segundo hilo hace lo contrario:

```
WaitHandle.SignalAndWait (wh2, wh1);
```

Alternativas a WaitAll y SignalAndWait

WaitAll y SignalAndWait no se ejecutarán en un apartamento de subprocesos de un solo hilo. Afortunadamente, existen alternativas. En el caso de SignalAndWait, rara vez se necesitan sus semánticas de salto de cola: en nuestro ejemplo de encuentro, por ejemplo, sería válido simplemente llamar a Set en el primer wait handle y luego WaitOne en el otro, si los wait handles se usaran únicamente para ese encuentro. En la siguiente sección, exploraremos otra opción para implementar un encuentro de hilos.

En el caso de WaitAny y WaitAll, si no necesitas atomicidad, puedes usar el código que escribimos en la sección anterior para convertir los wait handles en tareas y luego usar Task.WhenAny y Task.WhenAll (Capítulo 14).

Si necesitas atomicidad, puedes tomar el enfoque de nivel más bajo para la señalización y escribir la lógica tú mismo con los métodos Wait y Pulse de Monitor. Describimos Wait y Pulse en detalle en http://albahari.com/threading.

La clase Barrier

La clase Barrier implementa una barrera de ejecución de hilos, permitiendo que muchos hilos se encuentren en un punto en el tiempo (no debe confundirse con Thread.MemoryBarrier).

La clase es muy rápida y eficiente, y se basa en los métodos Wait, Pulse y spinlocks.

Para usar esta clase:

1. Instanciala, especificando cuántos hilos participarán en el encuentro (puedes cambiar esto más tarde llamando a AddParticipants/RemoveParticipants).

2. Haz que cada hilo llame a SignalAndWait cuando quiera encontrarse.

Instantiar Barrier con un valor de 3 hace que SignalAndWait se bloquee hasta que se llame a ese método tres veces. Luego, vuelve a comenzar: llamar a SignalAndWait nuevamente se bloquea hasta que se llame otras tres veces. Esto mantiene cada hilo "en sincronía" con todos los demás hilos.

En el siguiente ejemplo, cada uno de los tres hilos escribe los números del 0 al 4 manteniendo el paso con los otros hilos:

```
var barrier = new Barrier (3);

new Thread (Speak).Start();
new Thread (Speak).Start();
new Thread (Speak).Start();

void Speak()
{
  for (int i = 0; i < 5; i++)
  {
    Console.Write (i + " ");
    barrier.SignalAndWait();
  }
}

OUTPUT:  0 0 0 1 1 1 2 2 2 3 3 3 4 4 4
```

Una característica realmente útil de Barrier es que también puedes especificar una acción posterior a la fase al construirla. Esto es un delegado que se ejecuta después de que SignalAndWait ha sido llamado n veces, pero antes de que los hilos sean desbloqueados (como se muestra en el área sombreada en la Figura 21-3). En nuestro ejemplo, si instanciamos nuestra barrera de la siguiente manera:

```
static Barrier _barrier = new Barrier (3, barrier => Console.WriteLine());
```

la salida es la siguiente:

```
0 0 0
1 1 1
2 2 2
3 3 3
4 4 4
```

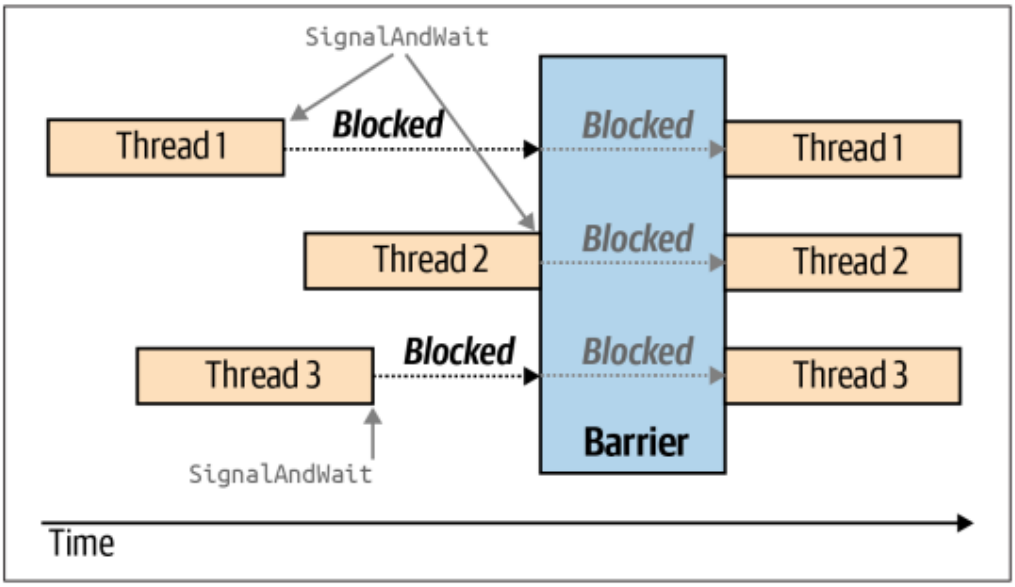


Figure 21-3. Barrier

Una acción de post-fase puede ser útil para fusionar datos de cada uno de los hilos de trabajo. No necesita preocuparse por la prelación, porque todos los trabajadores están bloqueados mientras realiza su tarea.

Inicialización perezosa

Un problema frecuente en el enhebrado es cómo inicializar perezosamente un campo compartido de manera segura para subprocesos. La necesidad surge cuando se tiene un campo de un tipo que es costoso de construir:

```
class Foo
{
  public readonly Expensive Expensive = new Expensive();
  ...
}
class Expensive {  /* Suppose this is expensive to construct */  }
```

El problema con este código es que instanciar `Foo` incurre en el costo de rendimiento de instanciar `Expensive`, sin importar si el campo `Expensive` se accede alguna vez.

La respuesta obvia es construir la instancia bajo demanda:

```
class Foo
{
  Expensive _expensive;
  public Expensive Expensive         // Lazily instantiate Expensive
  {
    get
    {
      if (_expensive == null) _expensive = new Expensive();
      return _expensive;
    }
  }
  ...
}
```

Entonces, surge la pregunta, ¿es esto seguro para subprocesos? Además del hecho de que estamos accediendo a `_expensive` fuera de un bloqueo sin una barrera de memoria, consideremos qué sucedería si dos hilos accedieran a esta propiedad al mismo tiempo. Ambos podrían cumplir el predicado del condicional `if` y cada hilo obtendría una instancia diferente de `Expensive`. Debido a que esto puede conducir a errores sutiles, en general, diríamos que este código no es seguro para subprocesos.

La solución al problema es bloquear alrededor de la verificación e inicialización del objeto:

```
Expensive _expensive;
readonly object _expenseLock = new object();

public Expensive Expensive
{
  get
  {
    lock (_expenseLock)
    {


      if (_expensive == null) _expensive = new Expensive();
      return _expensive;
    }
  }
}
```

Lazy<T>

La clase Lazy<T> está disponible para ayudar con la inicialización perezosa. Si se instancia con un argumento de true, implementa el patrón de inicialización segura para subprocesos descrito anteriormente.



Lazy<T> implementa en realidad una versión microoptimizada de este patrón, llamada bloqueo de doble comprobación. El bloqueo de doble comprobación realiza una lectura volátil adicional para evitar el costo de obtener un bloqueo si el objeto ya está inicializado.

Para usar Lazy<T>, se instancia la clase con un delegado de fábrica de valores que indica cómo inicializar un nuevo valor y el argumento true. Luego, se accede a su valor a través de la propiedad Value:

```
Lazy<Expensive> _expensive = new Lazy<Expensive>
  (() => new Expensive(), true);

public Expensive Expensive { get { return _expensive.Value; } }
```

Si se pasa false al constructor de Lazy<T>, implementa el patrón de inicialización perezosa no seguro para subprocesos que describimos al principio de esta sección, lo cual tiene sentido cuando se quiere usar Lazy<T> en un contexto de un solo subproceso.

LazyInitializer

LazyInitializer es una clase estática que funciona exactamente como Lazy<T> excepto por dos diferencias:

- Su funcionalidad se expone a través de un método estático que opera directamente en un campo de tu propio tipo. Esto evita un nivel de indirección, mejorando el rendimiento en casos donde se necesita una optimización extrema.
- Ofrece otro modo de inicialización en el cual varios subprocesos pueden competir por la inicialización.

Para usar LazyInitializer, se llama a EnsureInitialized antes de acceder al campo, pasando una referencia al campo y el delegado de fábrica:

```
Expensive _expensive;
public Expensive Expensive
{
  get          // Implement double-checked locking
  {
    LazyInitializer.EnsureInitialized (ref _expensive,
                                       () => new Expensive());
    return _expensive;
  }
}
```

Otra forma de inicializar la competencia entre los hilos para inicializar un objeto es utilizando un argumento adicional. Esto es similar a nuestro ejemplo original de no seguridad de hilos, excepto que siempre gana el primer hilo que termina, lo que resulta en una única instancia. La ventaja de esta técnica es que es aún más rápida (en sistemas multicore) que el bloqueo de doble comprobación, ya que se puede implementar completamente sin utilizar bloqueos utilizando técnicas avanzadas que se describen en "Sincronización no bloqueante" e "Inicialización diferida"

en http://albahari.com/threading. Esta es una optimización extrema (y raramente necesaria) que tiene un costo:

- Es más lenta cuando hay más hilos compitiendo para inicializar que el número de núcleos disponibles.
- Potencialmente desperdicia recursos de CPU al realizar inicializaciones redundantes.
- La lógica de inicialización debe ser segura para hilos (en este caso, sería insegura para hilos si el constructor de Expensive escribiera en campos estáticos, por ejemplo).
- Si el inicializador instancia un objeto que requiere eliminación, el objeto "desperdiciado" no se eliminará sin lógica adicional.

Almacenamiento local por hilo

Gran parte de este capítulo se ha centrado en las construcciones de sincronización y los problemas que surgen al tener hilos accediendo simultáneamente a los mismos datos. Sin embargo, a veces deseamos mantener los datos aislados, asegurando que cada hilo tenga su propia copia. Las variables locales logran precisamente esto, pero solo son útiles con datos transitorios.

La solución es el almacenamiento local por hilo. Podría resultarte difícil pensar en un caso de uso, ya que los datos que deseas mantener aislados en un hilo tienden a ser transitorios por naturaleza. Su principal aplicación es para almacenar datos "fuera de banda" que respaldan la infraestructura de la ruta de ejecución, como tokens de mensajería, transacciones y seguridad. Pasar estos datos como parámetros de método puede resultar incómodo y limitado a tus propios métodos; almacenar esta información en campos estáticos ordinarios significa compartirla entre todos los hilos.

El almacenamiento local por hilo también puede ser útil para optimizar el código paralelo. Permite que cada hilo acceda exclusivamente a su propia versión de un objeto que no es seguro para hilos sin necesidad de utilizar bloqueos y sin necesidad de reconstruir ese objeto entre llamadas de método.

Existen cuatro formas de implementar el almacenamiento local por hilo. Las examinaremos en las siguientes subsecciones.

[ThreadStatic]

El enfoque más sencillo para el almacenamiento local por hilo es marcar un campo estático con el atributo ThreadStatic:

```
[ThreadStatic] static int _x;
```

Cada hilo ve entonces una copia separada de _x.

Desafortunadamente, [ThreadStatic] no funciona con campos de instancia (simplemente no hace nada); tampoco funciona bien con inicializadores de campos, ya que se ejecutan solo una vez en el hilo que se está ejecutando cuando se ejecuta el constructor estático. Si necesitas

trabajar con campos de instancia o comenzar con un valor no predeterminado, ThreadLocal<T> proporciona una mejor opción.

ThreadLocal<T>

ThreadLocal<T> proporciona almacenamiento local por hilo tanto para campos estáticos como de instancia, y te permite especificar valores predeterminados.

Así es como se crea un ThreadLocal<int> con un valor predeterminado de 3 para cada hilo:

```
static ThreadLocal<int> _x = new ThreadLocal<int> (() => 3);
```

Luego, utilizas la propiedad Value de _x para obtener o establecer su valor local por hilo. Una ventaja de usar ThreadLocal es que los valores se evalúan de forma perezosa: la función de fábrica se evalúa en la primera llamada (para cada hilo).

ThreadLocal<T> y campos de instancia

ThreadLocal<T> también es útil con campos de instancia y variables locales capturadas. Por ejemplo, considera el problema de generar números aleatorios en un entorno multihilo. La clase Random no es segura para subprocesos, por lo que tenemos que bloquear su uso (limitando la concurrencia) o generar un objeto Random separado para cada hilo. ThreadLocal<T> facilita esta última opción:

```
var localRandom = new ThreadLocal<Random>(() => new Random());
Console.WriteLine (localRandom.Value.Next());
```

Sin embargo, nuestra función de fábrica para crear el objeto Random es un poco simplista, ya que el constructor sin parámetros de Random se basa en el reloj del sistema para generar una semilla de número aleatorio. Esto puede ser el mismo para dos objetos Random creados en ~10 ms uno después del otro. Aquí tienes una forma de solucionarlo:

```
var localRandom = new ThreadLocal<Random>
 ( () => new Random (Guid.NewGuid().GetHashCode()) );
```

Utilizamos esto en el Capítulo 22 (consulta el ejemplo de verificación ortográfica en paralelo en "PLINQ" en la página 911).

GetData y SetData

El tercer enfoque es utilizar dos métodos en la clase Thread: GetData y SetData. Estos almacenan datos en "compartimentos" específicos del hilo. Thread.GetData lee desde el almacén de datos aislados del hilo y Thread.SetData escribe en él. Ambos métodos requieren un objeto LocalDataStoreSlot para identificar el compartimento. Puedes utilizar el mismo compartimento en todos los hilos y aún así obtendrán valores separados. Aquí tienes un ejemplo:

```
class Test
{
  // The same LocalDataStoreSlot object can be used across all threads.
  LocalDataStoreSlot _secSlot = Thread.GetNamedDataSlot ("securityLevel");

  // This property has a separate value on each thread.
  int SecurityLevel
  {
    get
    {
      object data = Thread.GetData (_secSlot);
      return data == null ? 0 : (int) data;    // null == uninitialized
    }
    set { Thread.SetData (_secSlot, value); }
  }
  ...
```

En este ejemplo, llamamos a Thread.GetNamedDataSlot, que crea un compartimento con nombre, lo cual permite compartir ese compartimento en toda la aplicación. Alternativamente, puedes controlar el alcance de un compartimento con un compartimento sin nombre, obtenido mediante la llamada a Thread.AllocateDataSlot:

```
class Test
{
  LocalDataStoreSlot _secSlot = Thread.AllocateDataSlot();
  ...
```

Thread.FreeNamedDataSlot liberará un compartimento de datos con nombre en todos los hilos, pero solo una vez que todas las referencias a ese LocalDataStoreSlot hayan salido del ámbito y se hayan recogido mediante la recolección de basura. Esto garantiza que los hilos no tengan los compartimentos de datos retirados mientras los necesiten, siempre y cuando mantengan una referencia al objeto LocalDataStoreSlot correspondiente mientras se necesite el compartimento.

AsyncLocal<T>

Los enfoques de almacenamiento local por hilo que hemos discutido hasta ahora no son compatibles con funciones asíncronas, porque después de una espera (await), la ejecución puede continuar en otro hilo. La clase AsyncLocal<T> resuelve esto al preservar su valor a través de una espera:

```
static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();

async void Main()
{
  _asyncLocalTest.Value = "test";
  await Task.Delay (1000);
  // The following works even if we come back on another thread:
  Console.WriteLine (_asyncLocalTest.Value);   // test
}
```

AsyncLocal<T> aún puede mantener separadas las operaciones iniciadas en hilos diferentes, ya sea iniciadas por Thread.Start o Task.Run. Lo siguiente escribe "uno uno" y "dos dos":

```
static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();

void Main()
{
  // Call Test twice on two concurrent threads:
  new Thread (() => Test ("one")).Start();
  new Thread (() => Test ("two")).Start();
}

async void Test (string value)
{
  _asyncLocalTest.Value = value;
  await Task.Delay (1000);
  Console.WriteLine (value + " " + _asyncLocalTest.Value);
}
```

AsyncLocal<T> tiene una característica interesante y única: si un objeto AsyncLocal<T> ya tiene un valor cuando se inicia un hilo, el nuevo hilo "heredará" ese valor:

```
static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();

void Main()
{
  _asyncLocalTest.Value = "test";
  new Thread (AnotherMethod).Start();
}

void AnotherMethod() => Console.WriteLine (_asyncLocalTest.Value);  // test
```

Sin embargo, el nuevo hilo obtiene una copia del valor, por lo que cualquier cambio que realice no afectará al original:

```
static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();

void Main()
{
  _asyncLocalTest.Value = "test";
  var t = new Thread (AnotherMethod);
  t.Start(); t.Join();
  Console.WriteLine (_asyncLocalTest.Value);   // test  (not ha-ha!)
}

void AnotherMethod() => _asyncLocalTest.Value = "ha-ha!";
```

Ten en cuenta que el nuevo hilo obtiene una copia superficial del valor. Por lo tanto, si reemplazas Async<string> con Async<StringBuilder> o Async<List<string>>, el nuevo hilo podría borrar el StringBuilder o agregar/eliminar elementos de la List<string>, y esto afectaría al original.

Temporizadores (Timers)

Si necesitas ejecutar un método repetidamente a intervalos regulares, la forma más sencilla es utilizando un temporizador (timer). Los temporizadores son convenientes y eficientes en el uso de memoria y recursos, en comparación con técnicas como las siguientes:

```
new Thread (delegate() {
                          while (enabled)
                          {
                            DoSomeAction();
                            Thread.Sleep (TimeSpan.FromHours (24));
                          }
                        }).Start();
```

No solo esto ocupa permanentemente un recurso de hilo, sino que sin código adicional, DoSomeAction ocurrirá más tarde cada día. Los temporizadores resuelven estos problemas.

.NET proporciona cinco temporizadores. Dos de ellos son temporizadores multihilo de propósito general:

- System.Threading.Timer
- System.Timers.Timer

Los otros dos son temporizadores de un solo hilo de propósito especial:

- System.Windows.Forms.Timer (temporizador de Windows Forms)
- System.Windows.Threading.DispatcherTimer (temporizador de WPF)

Los temporizadores multihilo son más potentes, precisos y flexibles, mientras que los temporizadores de un solo hilo son más seguros y convenientes para ejecutar tareas simples que actualizan controles de Windows Forms o elementos de WPF.

Finalmente, a partir de .NET 6, está el PeriodicTimer, que cubriremos primero.

PeriodicTimer

PeriodicTimer no es realmente un temporizador, es una clase que ayuda con el bucle asincrónico. Es importante tener en cuenta que, desde la llegada de async y await, los temporizadores tradicionales generalmente no son necesarios. En su lugar, el siguiente patrón funciona bien:

```
StartPeriodicOperation();

async void StartPeriodicOperation()
{
  while (true)
  {
    await Task.Delay (1000);
    Console.WriteLine ("Tick");   // Do some action
  }
 }
```



Si llamas a StartPeriodicOperation desde un hilo de interfaz de usuario (UI), se comportará como un temporizador de un solo hilo, ya que el await siempre regresará en el mismo contexto de sincronización.

Puedes hacer que se comporte como un temporizador multihilo simplemente agregando .ConfigureAwait(false) al await.

El PeriodicTimer es una clase que simplifica este patrón:

```
var timer = new PeriodicTimer (TimeSpan.FromSeconds (1));
StartPeriodicOperation();
// Optionally dispose timer when you want to stop looping.

async void StartPeriodicOperation()
{
  while (await timer.WaitForNextTickAsync())
    Console.WriteLine ("Tick");    // Do some action
}
```

El PeriodicTimer también te permite detener el temporizador al deshacer la instancia del temporizador. Esto hace que WaitForNextTickAsync devuelva false, lo que permite que el bucle termine.

Temporizadores multihilo

System.Threading.Timer es el temporizador multihilo más simple: solo tiene un constructor y dos métodos (¡una delicia para minimalistas y también para los autores de libros!). En el siguiente ejemplo, un temporizador llama al método Tick, que escribe "tick..." después de que hayan transcurrido cinco segundos y luego cada segundo después de eso, hasta que el usuario presione Enter:

```
using System;
using System.Threading;

// First interval = 5000ms; subsequent intervals = 1000ms
Timer tmr = new Timer (Tick, "tick...", 5000, 1000);
Console.ReadLine();
tmr.Dispose();         // This both stops the timer and cleans up.

void Tick (object data)
{
  // This runs on a pooled thread
  Console.WriteLine (data);          // Writes "tick..."
}
```



Consulte "Temporizadores" en la página 577 para obtener una discusión sobre cómo desechar temporizadores multihilo.

Puedes cambiar el intervalo de un temporizador más adelante llamando a su método Change. Si deseas que el temporizador se active solo una vez, especifica Timeout.Infinite en el último argumento del constructor.

.NET también proporciona otra clase de temporizador con el mismo nombre en el espacio de nombres System.Timers. Esta simplemente envuelve System.Threading.Timer, proporcionando

conveniencia adicional mientras utiliza el mismo motor subyacente. Aquí hay un resumen de sus características añadidas:

- Implementación de IComponent, lo que le permite ser ubicado en la bandeja de componentes del diseñador de Visual Studio.
- Propiedad Interval en lugar de un método Change.
- Evento Elapsed en lugar de un delegado de devolución de llamada.
- Una propiedad Enabled para iniciar y detener el temporizador (su valor predeterminado es false).
- Los métodos Start y Stop en caso de que estés confundido por Enabled.
- Un indicador AutoReset para indicar un evento recurrente (su valor predeterminado es true).
- Una propiedad SynchronizingObject con los métodos Invoke y BeginInvoke para llamar de manera segura a métodos en elementos de WPF y controles de Windows Forms.

Aquí tienes un ejemplo:

```
using System;
using System.Timers;          // Timers namespace rather than Threading

var tmr = new Timer();        // Doesn't require any args
tmr.Interval = 500;
tmr.Elapsed += tmr_Elapsed;   // Uses an event instead of a delegate
tmr.Start();                  // Start the timer
Console.ReadLine();
tmr.Stop();                   // Stop the timer
Console.ReadLine();
tmr.Start();                  // Restart the timer
Console.ReadLine();
tmr.Dispose();                // Permanently stop the timer

void tmr_Elapsed (object sender, EventArgs e)
  => Console.WriteLine ("Tick");
```

Los temporizadores multihilo utilizan el grupo de hilos (thread pool) para permitir que unos pocos hilos sirvan a muchos temporizadores. Esto significa que el método de devolución de llamada (callback) o el evento Elapsed pueden ejecutarse en un hilo diferente cada vez que se llamen. Además, el evento Elapsed siempre se dispara aproximadamente a tiempo, independientemente de si el evento Elapsed anterior terminó su ejecución. Por lo tanto, los métodos de devolución de llamada o los controladores de eventos deben ser seguros para subprocesos (thread-safe).

La precisión de los temporizadores multihilo depende del sistema operativo y generalmente está en el rango de 10 a 20 milisegundos. Si necesitas una mayor precisión, puedes utilizar la interoperabilidad nativa y llamar al temporizador multimedia de Windows. Esto tiene una precisión de hasta un milisegundo y está definido en winmm.dll. Primero llama a timeBeginPeriod para informar al sistema operativo que necesitas una alta precisión en el temporizador, luego llama a timeSetEvent para iniciar el temporizador multimedia. Cuando hayas terminado, llama a timeKillEvent para detener el temporizador y a timeEndPeriod para informar al sistema operativo que ya no necesitas una alta precisión en el tiempo.

El Capítulo 24 muestra cómo llamar a métodos externos con P/Invoke. Puedes encontrar ejemplos completos en internet que utilizan el temporizador multimedia buscando las palabras clave "dllimport winmm.dll timesetevent".

Temporizadores de un solo hilo

.NET proporciona temporizadores diseñados para eliminar problemas de seguridad de subprocesos (thread-safety) en aplicaciones WPF y Windows Forms:

- System.Windows.Threading.DispatcherTimer (WPF)
- System.Windows.Forms.Timer (Windows Forms)



Los temporizadores de un solo hilo no están diseñados para funcionar fuera de sus respectivos entornos. Por ejemplo, si utilizas un temporizador de Windows Forms en una aplicación de servicio de Windows, ¡el evento del temporizador no se activará!

Ambos son similares a System.Timers.Timer en los miembros que exponen, como Interval, Start y Stop (y Tick, que es equivalente a Elapsed), y se utilizan de manera similar. Sin embargo, difieren en cómo funcionan internamente. En lugar de disparar eventos de temporizador en hilos del grupo de hilos (thread pool), envían los eventos al bucle de mensajes de WPF o Windows Forms. Esto significa que el evento Tick siempre se activa en el mismo hilo que creó originalmente el temporizador, que en una aplicación normal es el mismo hilo utilizado para administrar todos los elementos y controles de la interfaz de usuario. Esto tiene varios beneficios:

- Puedes olvidarte de la seguridad de subprocesos (thread safety).
- Un nuevo Tick nunca se activará hasta que el Tick anterior haya terminado de procesarse.
- Puedes actualizar elementos y controles de la interfaz de usuario directamente desde el código de manejo del evento Tick sin llamar a Control.BeginInvoke o Dispatcher.BeginInvoke.

Por lo tanto, un programa que utiliza estos temporizadores no es realmente multihilo: terminas con el mismo tipo de pseudoconcurrencia que se describe en el Capítulo 14 con funciones asíncronas que se ejecutan en un hilo de interfaz de usuario (UI). Un solo hilo sirve a todos los temporizadores, así como al procesamiento de eventos de la interfaz de usuario, lo que significa que el controlador de eventos Tick debe ejecutarse rápidamente, de lo contrario, la interfaz de usuario se vuelve no receptiva.

Esto hace que los temporizadores de WPF y Windows Forms sean adecuados para tareas pequeñas, como actualizar algún aspecto de la interfaz de usuario (por ejemplo, un reloj o una pantalla de cuenta regresiva).

En cuanto a la precisión, los temporizadores de un solo hilo son similares a los temporizadores multihilo (tens de milisegundos), aunque generalmente son menos precisos porque pueden retrasarse mientras se procesan otras solicitudes de la interfaz de usuario (u otros eventos de temporizador).

22 Programación Paralela (Parallel Programming)

En este capítulo, cubrimos las API y constructos de programación multihilo destinados a aprovechar los procesadores multinúcleo:

- Parallel LINQ, o PLINQ
- La clase Parallel
- Los constructos de paralelismo de tareas
- Las colecciones concurrentes

Estos constructos se conocen colectivamente (de manera general) como Parallel Framework (PFX). La clase Parallel junto con los constructos de paralelismo de tareas se denomina Task Parallel Library (TPL).

Debes sentirte cómodo con los fundamentos en el Capítulo 14 antes de leer este capítulo, especialmente con el bloqueo, la seguridad de subprocesos y la clase Task.



.NET ofrece una serie de API especializadas adicionales para ayudar con la programación paralela y asíncrona:

- System.Threading.Channels.Channel es una cola de productor/consumidor asincrónica de alto rendimiento introducida en .NET Core 3.
- Microsoft Dataflow (en el espacio de nombres System.Threading.Tasks.Dataflow) es una API sofisticada para crear redes de bloques en búfer que ejecutan acciones o transformaciones de datos en paralelo, con una semejanza a la programación de actores/agentes.
- Reactive Extensions implementa LINQ sobre IObservable (una abstracción alternativa a IAsyncEnumerable) y sobresale en la combinación de flujos asíncronos. Reactive Extensions se encuentra en el paquete NuGet System.Reactive.

¿Por qué PFX?

En los últimos 15 años, los fabricantes de CPU han pasado de procesadores de un solo núcleo a procesadores multinúcleo. Esto es problemático para nosotros como programadores porque el código de un solo hilo no se ejecuta automáticamente más rápido como resultado de esos núcleos adicionales.

Aprovechar múltiples núcleos es fácil para la mayoría de las aplicaciones de servidor, donde cada hilo puede manejar de forma independiente una solicitud de cliente separada, pero es más difícil en el escritorio porque generalmente requiere que tomes tu código intensivo en computación y hagas lo siguiente:

1. Dividirlo en pequeños fragmentos.

2. Ejecutar esos fragmentos en paralelo mediante el uso de subprocesos múltiples.

3. Recopilar los resultados a medida que estén disponibles, de manera segura y eficiente en términos de rendimiento.

Aunque puedes hacer todo esto con los constructos clásicos de multihilo, resulta incómodo, especialmente los pasos de particionar y recopilar. Otro problema es que la estrategia habitual de bloqueo para la seguridad de subprocesos genera mucha contención cuando muchos subprocesos trabajan en los mismos datos al mismo tiempo.

Las bibliotecas PFX han sido diseñadas específicamente para ayudar en estos escenarios.



Programar para aprovechar múltiples núcleos o múltiples procesadores se denomina programación paralela. Esto es un subconjunto del concepto más amplio de multihilo.

Conceptos de PFX

Existen dos estrategias para particionar el trabajo entre subprocesos: paralelismo de datos y paralelismo de tareas.

Cuando se deben realizar un conjunto de tareas en muchos valores de datos, podemos paralelizar al hacer que cada subproceso realice el (mismo) conjunto de tareas en un subconjunto de valores. Esto se denomina paralelismo de datos porque estamos dividiendo los datos entre subprocesos. En contraste, con el paralelismo de tareas, particionamos las tareas; es decir, cada subproceso realiza una tarea diferente.

En general, el paralelismo de datos es más fácil y escala mejor en hardware altamente paralelo porque reduce o elimina los datos compartidos (lo que reduce la contención y los problemas de seguridad de subprocesos). Además, el paralelismo de datos aprovecha el hecho de que a menudo hay más valores de datos que tareas discretas, lo que aumenta el potencial de paralelismo.

El paralelismo de datos también es propicio para el paralelismo estructurado, lo que significa que las unidades de trabajo paralelas comienzan y terminan en el mismo lugar de tu programa. En contraste, el paralelismo de tareas tiende a ser no estructurado, lo que significa que las unidades de trabajo paralelas pueden comenzar y terminar en lugares dispersos en tu programa. El paralelismo estructurado es más simple y menos propenso a errores, y te permite delegar el trabajo difícil de particionar y coordinar los subprocesos (e incluso la recopilación de resultados) a bibliotecas.

Componentes de PFX

PFX consta de dos capas de funcionalidad, como se muestra en la Figura 22-1. La capa superior consta de dos API estructuradas de paralelismo de datos: PLINQ y la clase Parallel. La capa inferior contiene las clases de paralelismo de tareas, además de un conjunto de construcciones adicionales que ayudan con las actividades de programación paralela.

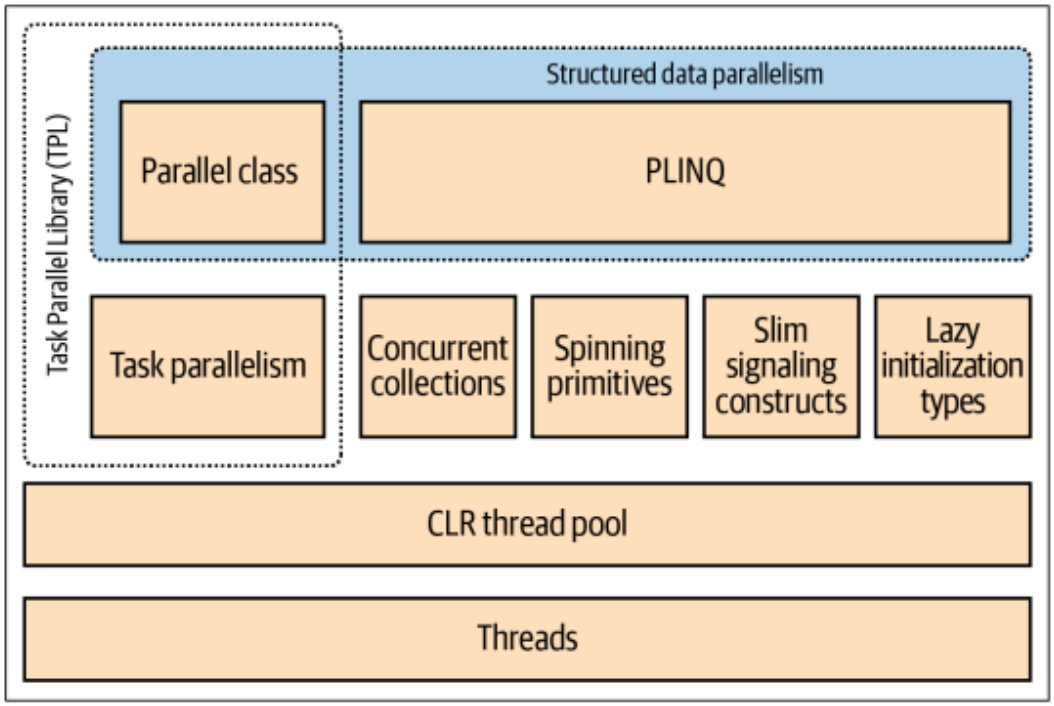


Figure 22-1. PFX components

PLINQ ofrece la funcionalidad más completa: automatiza todos los pasos de paralelización, incluyendo la partición del trabajo en tareas, la ejecución de esas tareas en hilos y la recopilación de los resultados en una sola secuencia de salida. Se le llama declarativo porque simplemente declaras que quieres paralelizar tu trabajo (que estructuras como una consulta LINQ) y permites que el tiempo de ejecución se encargue de los detalles de implementación.

En contraste, los otros enfoques son imperativos, ya que debes escribir explícitamente código para la partición o recopilación. Como muestra el siguiente resumen, en el caso de la clase Parallel, debes recopilar los resultados tú mismo; con las construcciones de paralelismo de tareas, también debes particionar el trabajo tú mismo:

	Partitions work	Collates results
PLINQ	Yes	Yes
The `Parallel` class	Yes	No
PFX's task parallelism	No	No

Las colecciones concurrentes y las primitivas de spinning te ayudan con actividades de programación paralela a nivel más bajo. Estas son importantes porque PFX ha sido diseñado para funcionar no solo con el hardware actual, sino también con generaciones futuras de procesadores con muchos más núcleos. Si quieres mover un montón de madera cortada y tienes 32 trabajadores para hacer el trabajo, el mayor desafío es mover la madera sin que los trabajadores se estorben entre sí. Lo mismo ocurre al dividir un algoritmo entre 32 núcleos: si se utilizan cerraduras ordinarias para proteger recursos comunes, el bloqueo resultante puede hacer que solo una fracción de esos núcleos esté realmente ocupada a la vez. Las colecciones concurrentes están optimizadas específicamente para el acceso altamente concurrente, con el enfoque de minimizar o eliminar el bloqueo. PLINQ y la clase Parallel se basan en las colecciones concurrentes y en las primitivas de spinning para una gestión eficiente del trabajo.

Cuándo utilizar PFX

El caso de uso principal para PFX es la programación paralela: aprovechar los procesadores multinúcleo para acelerar el código intensivo en cómputo.

Un desafío en la programación paralela es la ley de Amdahl, que establece que la mejora de rendimiento máxima obtenida mediante la paralelización está gobernada por la porción del código que debe ejecutarse secuencialmente. Por ejemplo, si solo dos tercios del tiempo de ejecución de un algoritmo son paralelizables, nunca podrás superar una mejora de rendimiento de tres veces, incluso con un número infinito de núcleos.

Otras aplicaciones de PFX

Las construcciones de programación paralela son útiles no solo para aprovechar los múltiples núcleos, sino también en otros escenarios:

• Las colecciones concurrentes son apropiadas en ocasiones cuando se necesita una cola, pila o diccionario seguros para subprocesos.

• BlockingCollection proporciona una forma sencilla de implementar estructuras de productor/consumidor y es una buena manera de limitar la concurrencia.

• Las tareas son la base de la programación asíncrona, como vimos en el Capítulo 14.

Antes de continuar, es importante verificar si el cuello de botella está en el código paralelizable. También es importante considerar si su código necesita ser intensivo en cómputo, ya que la optimización a menudo es el enfoque más fácil y efectivo. Sin embargo, hay un compromiso, ya que algunas técnicas de optimización pueden dificultar la paralelización del código.

Las ganancias más fáciles se obtienen con lo que se llama problemas embarazosamente paralelos, que es cuando un trabajo se puede dividir fácilmente en tareas que se ejecutan de manera eficiente por sí mismas (el paralelismo estructurado es muy adecuado para este tipo de problemas). Ejemplos de esto incluyen muchas tareas de procesamiento de imágenes, trazado de rayos y enfoques de fuerza bruta en matemáticas o criptografía. Un ejemplo de un problema no embarazosamente paralelo es la implementación de una versión optimizada del algoritmo de ordenación quicksort, donde obtener un buen resultado requiere un poco de reflexión y puede requerir paralelismo no estructurado.

PLINQ

PLINQ (Parallel LINQ) es una extensión de LINQ que permite la paralelización automática de consultas LINQ locales. PLINQ tiene la ventaja de ser fácil de usar, ya que se encarga tanto de la partición del trabajo como de la recopilación de resultados en .NET.

Para utilizar PLINQ, simplemente llama a AsParallel() en la secuencia de entrada y luego continúa la consulta LINQ como de costumbre. La siguiente consulta calcula los números primos entre 3 y 100,000, aprovechando al máximo todos los núcleos en la máquina de destino:

```
// Calculate prime numbers using a simple (unoptimized) algorithm.

IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);

var parallelQuery =
  from n in numbers.AsParallel()
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;

int[] primes = parallelQuery.ToArray();
```

AsParallel es un método de extensión en System.Linq.ParallelEnumerable. Envuelve la secuencia de entrada en una secuencia basada en ParallelQuery<TSource>, lo que hace que los operadores de consulta LINQ que posteriormente llames se enlacen con un conjunto alternativo de métodos de extensión definidos en ParallelEnumerable. Estos proporcionan implementaciones paralelas de cada uno de los operadores de consulta estándar. Básicamente, funcionan mediante la partición de la secuencia de entrada en fragmentos que se ejecutan en diferentes hilos, recopilando los resultados en una única secuencia de salida para su consumo, como se muestra en la Figura 22-2.

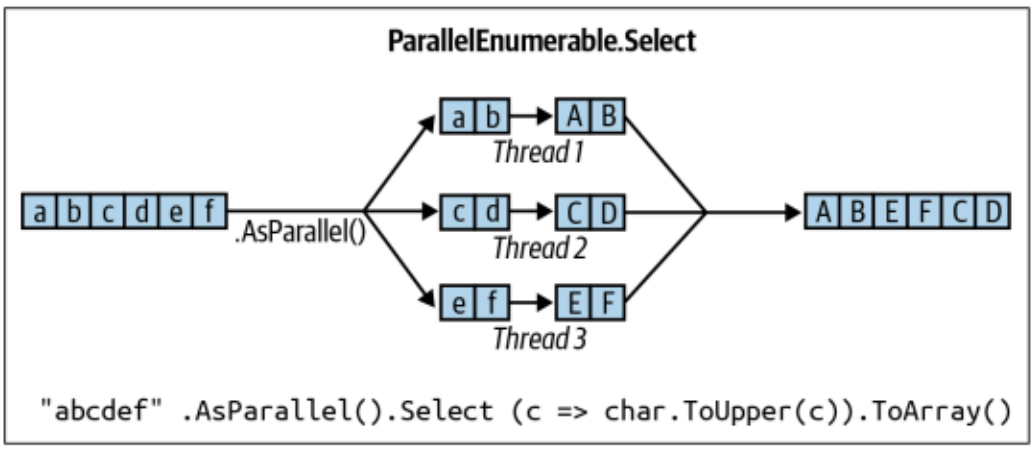


Figure 22-2. PLINQ execution model

Llamar a AsSequential() deshace una secuencia de ParallelQuery para que los operadores de consulta subsiguientes se vinculen a los operadores de consulta estándar y se ejecuten de forma secuencial. Esto es necesario antes de llamar a métodos que tengan efectos secundarios o no sean seguros en entornos multiproceso.

Para los operadores de consulta que aceptan dos secuencias de entrada (Join, GroupJoin, Concat, Union, Intersect, Except y Zip), debes aplicar AsParallel() a ambas secuencias de entrada (de lo contrario, se producirá una excepción). Sin embargo, no es necesario seguir aplicando AsParallel a una consulta a medida que avanza, ya que los operadores de consulta de PLINQ generan otra secuencia de ParallelQuery. De hecho, llamar a AsParallel nuevamente introduce ineficiencia ya que obliga a fusionar y volver a particionar la consulta.

```
mySequence.AsParallel()          // Wraps sequence in ParallelQuery<int>
          .Where (n => n > 100)  // Outputs another ParallelQuery<int>
          .AsParallel()          // Unnecessary - and inefficient!
          .Select (n => n * n)
```

No todos los operadores de consulta se pueden paralelizar de manera efectiva. Para aquellos que no se pueden paralelizar (ver "Limitaciones de PLINQ" en la página 914), PLINQ implementa el operador de forma secuencial. PLINQ también puede operar de manera secuencial si sospecha que el costo de la paralelización ralentizará una consulta en particular.

PLINQ solo se aplica a colecciones locales y no funciona con Entity Framework, por ejemplo, porque en esos casos, LINQ se traduce a SQL, que luego se ejecuta en un servidor de base de datos. Sin embargo, puedes usar PLINQ para realizar consultas locales adicionales en los conjuntos de resultados obtenidos de consultas de base de datos.



Si una consulta de PLINQ genera una excepción, se lanza como una AggregateException cuya propiedad InnerExceptions contiene la verdadera excepción (o excepciones). Para obtener más detalles, consulta "Trabajando con AggregateException" en la página 940.

Ejecución paralela balística (Parallel Execution Ballistics)

Al igual que las consultas LINQ ordinarias, las consultas PLINQ se evalúan de forma diferida. Esto significa que la ejecución se activa solo cuando comienzas a consumir los resultados, generalmente a través de un bucle foreach (aunque también puede ser a través de un operador de conversión como ToArray o un operador que devuelve un solo elemento o valor).

Sin embargo, al enumerar los resultados, la ejecución se realiza de manera algo diferente a una consulta secuencial ordinaria. Una consulta secuencial es impulsada completamente por el consumidor de manera "push": cada elemento de la secuencia de entrada se recupera exactamente cuando es necesario por el consumidor. Una consulta paralela normalmente utiliza hilos independientes para obtener elementos de la secuencia de entrada ligeramente antes de que sean necesarios por el consumidor (similar a un teleprompter para los presentadores de noticias). Luego procesa los elementos en paralelo a través de la cadena de consultas, manteniendo los resultados en un búfer pequeño para que estén listos para el consumidor cuando se soliciten. Si el consumidor pausa o interrumpe la enumeración temprano, el procesador de consultas también se pausa o detiene para no desperdiciar tiempo de CPU o memoria.



Puedes ajustar el comportamiento de almacenamiento en búfer de PLINQ llamando a WithMergeOptions después de AsParallel. El valor predeterminado de AutoBuffered generalmente brinda los mejores resultados generales. NotBuffered deshabilita el búfer y es útil si deseas ver resultados lo antes posible. FullyBuffered almacena en caché el conjunto completo de resultados antes de presentarlo al consumidor (los operadores OrderBy y Reverse funcionan

naturalmente de esta manera, al igual que los operadores de elementos, agregación y conversión).

¿Por qué AsParallel no es la opción predeterminada? (Why Isn't AsParallel the Default?)

La pregunta de por qué Microsoft no hizo de PLINQ la opción predeterminada y paralelizó automáticamente los operadores de consulta estándar tiene varias razones. En primer lugar, para que PLINQ sea útil, debe haber una cantidad razonable de trabajo intensivo en computación para que se pueda distribuir en hilos de trabajo. La mayoría de las consultas de LINQ a objetos se ejecutan muy rápidamente, por lo que la paralelización no sería necesaria y el sobrecosto de particionar, combinar y coordinar los hilos adicionales podría ralentizar las cosas.

Además:

- La salida de una consulta PLINQ puede diferir de una consulta LINQ en cuanto al orden de los elementos.

- PLINQ envuelve las excepciones en una AggregateException (para manejar la posibilidad de múltiples excepciones lanzadas).

- PLINQ puede producir resultados poco confiables si la consulta invoca métodos no seguros para subprocesos.

Por último, PLINQ ofrece varias opciones para ajustar y optimizar su funcionamiento. Incorporar todas estas sutilezas a la API estándar de LINQ a objetos podría generar distracciones innecesarias.

PLINQ y el ordenamiento

Un efecto secundario de paralelizar los operadores de consulta es que cuando se collation los resultados, no necesariamente se mantienen en el mismo orden en que se presentaron (ver Figura 22-2). En otras palabras, la garantía de preservación del orden de LINQ para secuencias ya no se cumple.

Si necesitas preservar el orden, puedes forzarlo llamando a AsOrdered() después de AsParallel():

```
myCollection.AsParallel().AsOrdered()...
```

Llamar a AsOrdered incurre en un costo de rendimiento con grandes cantidades de elementos porque PLINQ debe realizar un seguimiento de la posición original de cada elemento.

Puedes negar el efecto de AsOrdered más adelante en una consulta llamando a AsUnordered: esto introduce un "punto de mezcla aleatorio", que permite que la consulta se ejecute de manera más eficiente a partir de ese punto. Entonces, si deseas preservar el orden de la secuencia de entrada solo para los dos primeros operadores de consulta, harías lo siguiente:

```
inputSequence.AsParallel().AsOrdered()
  .QueryOperator1()
  .QueryOperator2()
  .AsUnordered()       // From here on, ordering doesn't matter
  .QueryOperator3()
  ...
```

AsOrdered no es la opción predeterminada porque para la mayoría de las consultas, el orden original de entrada no es importante. En otras palabras, si AsOrdered fuera la opción predeterminada, necesitarías aplicar AsUnordered a la mayoría de tus consultas paralelas para obtener el mejor rendimiento, lo cual sería una carga.

Limitaciones de PLINQ

Existen limitaciones prácticas en lo que PLINQ puede paralelizar. Los siguientes operadores de consulta evitan la paralelización de forma predeterminada a menos que los elementos de origen estén en su posición de indexación original:

The indexed versions of `Select`, `SelectMany`, and `ElementAt`

La mayoría de los operadores de consulta cambian la posición de indexación de los elementos (incluidos aquellos que eliminan elementos, como Where). Esto significa que si deseas utilizar los operadores anteriores, generalmente deben estar al comienzo de la consulta.

Los siguientes operadores de consulta son paralelizables, pero utilizan una estrategia costosa de particionamiento que a veces puede ser más lenta que el procesamiento secuencial:

`Join`, `GroupBy`, `GroupJoin`, `Distinct`, `Union`, `Intersect`, and `Except`

Las versiones sembradas del operador Aggregate en sus encarnaciones estándar no son paralelizables: PLINQ proporciona versiones especiales para manejar esto (ver "Optimización de PLINQ" en la página 919).

Todos los demás operadores son paralelizables, aunque el uso de estos operadores no garantiza que la consulta se paralelice. PLINQ puede ejecutar tu consulta de manera secuencial si sospecha que el costo de la paralelización ralentizará esa consulta en particular. Puedes anular este comportamiento y forzar la paralelización llamando a lo siguiente después de AsParallel():

```
.WithExecutionMode (ParallelExecutionMode.ForceParallelism)
```

Ejemplo: Corrector Ortográfico Paralelo

Supongamos que queremos escribir un corrector ortográfico que se ejecute rápidamente con documentos muy grandes utilizando todos los núcleos disponibles. Al formular nuestro algoritmo en una consulta LINQ, podemos paralelizarlo fácilmente.

El primer paso es descargar un diccionario de palabras en inglés en un HashSet para una búsqueda eficiente:

```
if (!File.Exists ("WordLookup.txt")    // Contains about 150,000 words
  File.WriteAllText ("WordLookup.txt",
    await new HttpClient().GetStringAsync (
      "http://www.albahari.com/ispell/allwords.txt"));

var wordLookup = new HashSet<string> (
  File.ReadAllLines ("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);
```

Luego, utilizamos nuestra búsqueda de palabras para crear un "documento" de prueba que consiste en un array de un millón de palabras aleatorias. Después de construir el array, introducimos un par de errores ortográficos:

```
var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();

wordsToTest [12345] = "woozsh";     // Introduce a couple
wordsToTest [23456] = "wubsie";     // of spelling mistakes.
```

Ahora podemos realizar nuestra corrección ortográfica paralela probando las palabras de wordsToTest contra wordLookup. PLINQ facilita esto:

```
var query = wordsToTest
  .AsParallel()
  .Select  ((word, index) => new IndexedWord { Word=word, Index=index })
  .Where   (iword => !wordLookup.Contains (iword.Word))
  .OrderBy (iword => iword.Index);

foreach (var mistake in query)
  Console.WriteLine (mistake.Word + " - index = " + mistake.Index);

// OUTPUT:
// woozsh - index = 12345
// wubsie - index = 23456
```

IndexedWord es una estructura personalizada que definimos de la siguiente manera:

```
struct IndexedWord { public string Word; public int Index; }
```

El método wordLookup.Contains en el predicado le da a la consulta cierta "sustancia" y la hace valiosa para paralelizar.



Podríamos simplificar ligeramente la consulta utilizando un tipo anónimo en lugar de la estructura IndexedWord. Sin embargo, esto degradaría el rendimiento porque los tipos anónimos (al ser clases y, por lo tanto, tipos de referencia) incurren en el costo de la asignación en el montón y la posterior recolección de basura.

La diferencia puede que no sea suficiente como para tener importancia en consultas secuenciales, pero en consultas paralelas, favorecer la asignación en la pila puede ser bastante ventajoso. Esto se debe a que la asignación en la pila es altamente paralelizable (ya que cada hilo tiene su propia pila), mientras que todos los hilos deben competir por el mismo montón, gestionado por un único administrador de memoria y recolector de basura.

Uso de ThreadLocal<T>

Ampliemos nuestro ejemplo al paralelizar la creación de la lista de palabras de prueba aleatorias. Lo estructuramos como una consulta LINQ, por lo que debería ser sencillo. Aquí está la versión secuencial:

```
string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();
```

Desafortunadamente, la llamada a random.Next no es segura para subprocesos, por lo que no es tan simple como insertar AsParallel() en la consulta. Una solución potencial es escribir una función que bloquee alrededor de random.Next; sin embargo, esto limitaría la concurrencia. La mejor opción es utilizar ThreadLocal<Random> (ver "Almacenamiento local por hilo" en la página 898) para crear un objeto Random separado para cada hilo. Luego, podemos paralelizar la consulta de la siguiente manera:

```
var localRandom = new ThreadLocal<Random>
 ( () => new Random (Guid.NewGuid().GetHashCode()) );

string[] wordsToTest = Enumerable.Range (0, 1000000).AsParallel()
  .Select (i => wordList [localRandom.Value.Next (0, wordList.Length)])
  .ToArray();
```

En nuestra función de fábrica para instanciar un objeto Random, pasamos el hashcode de un Guid para asegurarnos de que si se crean dos objetos Random en un corto período de tiempo, generarán secuencias de números aleatorios diferentes.

Cuándo usar PLINQ

Es tentador buscar en tus aplicaciones existentes consultas LINQ y experimentar con su paralelización. Sin embargo, esto suele ser improductivo, ya que la mayoría de los problemas para los cuales LINQ es claramente la mejor solución tienden a ejecutarse muy rápidamente y, por lo tanto, no se benefician de la paralelización. Un enfoque mejor es encontrar un cuello de botella intensivo en CPU y considerar si se puede expresar como una consulta LINQ. (Un efecto secundario positivo de esta reestructuración es que LINQ suele hacer que el código sea más pequeño y legible).

PLINQ es muy adecuado para problemas "embarazosamente paralelos". Sin embargo, puede ser una mala elección para el procesamiento de imágenes, ya que reunir millones de píxeles en una secuencia de salida crea un cuello de botella. En su lugar, es mejor escribir los píxeles directamente en un arreglo o bloque de memoria no administrada y utilizar la clase Parallel o el paralelismo de tareas para gestionar el multihilo. (Sin embargo, es posible evitar la reunión de

resultados utilizando ForAll, lo cual discutimos en "Optimización de PLINQ" en la página 919. Hacerlo tiene sentido si el algoritmo de procesamiento de imágenes se presta naturalmente a LINQ).

Pureza funcional

Debido a que PLINQ ejecuta tu consulta en hilos paralelos, debes tener cuidado de no realizar operaciones no seguras para los hilos. En particular, escribir en variables tiene efectos secundarios y, por lo tanto, no es seguro para los hilos:

```
// The following query multiplies each element by its position.
// Given an input of Enumerable.Range(0,999), it should output squares.
int i = 0;
var query = from n in Enumerable.Range(0,999).AsParallel() select n * i++;
```

Podríamos hacer que la incrementación de i sea segura para los hilos utilizando bloqueos, pero el problema seguiría siendo que i no necesariamente se corresponderá con la posición del elemento de entrada. Y agregar AsOrdered a la consulta no solucionaría el último problema, porque AsOrdered solo garantiza que los elementos se emitan en un orden coherente con el procesamiento secuencial, no los procesa realmente en secuencia.

La solución correcta es reescribir nuestra consulta para utilizar la versión indexada de Select:

```
var query = Enumerable.Range(0,999).AsParallel().Select ((n, i) => n * i);
```

Para obtener el mejor rendimiento, cualquier método llamado desde los operadores de consulta debería ser seguro para los hilos en virtud de no escribir en campos o propiedades (no tener efectos secundarios, o ser funcionalmente puro). Si son seguros para los hilos gracias al uso de bloqueos, el potencial de paralelismo de la consulta se verá limitado por la duración del bloqueo dividido por el tiempo total invertido en esa función.

Configuración del grado de paralelismo

Por defecto, PLINQ elige un grado de paralelismo óptimo para el procesador en uso. Puedes sobrescribirlo llamando a WithDegreeOfParallelism después de AsParallel:

```
...AsParallel().WithDegreeOfPallelism(4)...
```

Un ejemplo en el que podrías aumentar el paralelismo más allá del número de núcleos es cuando tienes trabajo orientado a E/S (por ejemplo, descargar varias páginas web al mismo tiempo). Sin embargo, los combinadores de tareas y las funciones asíncronas proporcionan una solución igualmente fácil y más eficiente (consultar "Combinadores de tareas" en la página 663). A diferencia de las tareas, PLINQ no puede realizar trabajo orientado a E/S sin bloquear hilos (y los hilos en grupo, lo que empeora las cosas).

Cambio del grado de paralelismo

Solo puedes llamar a WithDegreeOfParallelism una vez dentro de una consulta PLINQ. Si necesitas llamarlo nuevamente, debes forzar la fusión y repartición de la consulta llamando a AsParallel() nuevamente dentro de la consulta:

```
"The Quick Brown Fox"
  .AsParallel().WithDegreeOfParallelism (2)
  .Where (c => !char.IsWhiteSpace (c))
  .AsParallel().WithDegreeOfParallelism (3)   // Forces Merge + Partition
  .Select (c => char.ToUpper (c))
```

Cancelacion (Cancellation)

Cancelando una consulta PLINQ cuyos resultados estás consumiendo en un bucle foreach es fácil: simplemente rompe el bucle foreach y la consulta se cancelará automáticamente cuando el enumerador se cierre implícitamente.

Para una consulta que finaliza con un operador de conversión, elemento o agregación, puedes cancelarla desde otro hilo mediante un token de cancelación (consulta "Cancelación" en la página 659). Para insertar un token, llama a WithCancellation después de llamar a AsParallel, pasando la propiedad Token de un objeto CancellationTokenSource. Otro hilo puede llamar a Cancel en el origen del token, lo que lanzará una excepción OperationCanceledException en el consumidor de la consulta:

```
IEnumerable<int> million = Enumerable.Range (3, 1000000);

var cancelSource = new CancellationTokenSource();

var primeNumberQuery =
  from n in million.AsParallel().WithCancellation (cancelSource.Token)
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;

new Thread (() => {
                    Thread.Sleep (100);      // Cancel query after
                    cancelSource.Cancel();   // 100 milliseconds.
                  }
           ).Start();
try
{
  // Start query running:
  int[] primes = primeNumberQuery.ToArray();
  // We'll never get here because the other thread will cancel us.
}
catch (OperationCanceledException)
{
  Console.WriteLine ("Query canceled");
}
```

Al cancelar, PLINQ espera a que cada hilo de trabajo termine con su elemento actual antes de finalizar la consulta. Esto significa que cualquier método externo que la consulta llame se ejecutará hasta su finalización.

Optimizando PLINQ

Optimización de salida

Una de las ventajas de PLINQ es que permite juntar de manera conveniente los resultados del trabajo paralelizado en una única secuencia de salida. Sin embargo, a veces, todo lo que haces con esa secuencia es ejecutar una función una vez por cada elemento:

```
foreach (int n in parallelQuery)
  DoSomething (n);
```

Si este es el caso, y no te importa el orden en el que se procesan los elementos, puedes mejorar la eficiencia utilizando el método ForAll de PLINQ.

El método ForAll ejecuta un delegado en cada elemento de salida de una ParallelQuery. Se conecta directamente con las partes internas de PLINQ, evitando los pasos de juntar y enumerar los resultados. Aquí tienes un ejemplo trivial:

```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ForAll (Console.Write);
```

La Figura 22-3 muestra el proceso.



La operación de juntar y enumerar los resultados no es excesivamente costosa, por lo que la optimización con ForAll ofrece mayores beneficios cuando hay un gran número de elementos de entrada que se ejecutan rápidamente.

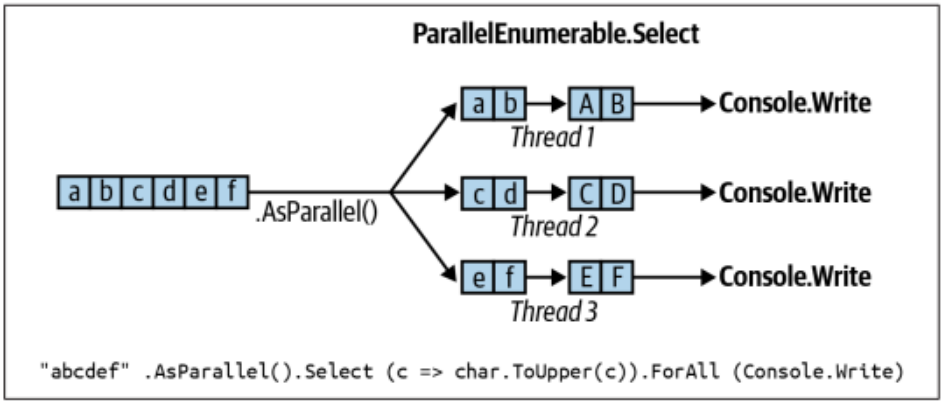


Figure 22-3. PLINQ ForAll

Optimización de entrada

PLINQ tiene tres estrategias de particionamiento para asignar elementos de entrada a hilos:

Strategy	Element allocation	Relative performance
Chunk partitioning	Dynamic	Average
Range partitioning	Static	Poor to excellent
Hash partitioning	Static	Poor

Para los operadores de consulta que requieren comparar elementos (GroupBy, Join, GroupJoin, Intersect, Except, Union y Distinct), no tienes opción: PLINQ siempre utiliza particionamiento por hash. El particionamiento por hash es relativamente ineficiente porque debe calcular previamente el código hash de cada elemento (para que los elementos con códigos hash idénticos se procesen en el mismo hilo). Si encuentras que esto es demasiado lento, tu única opción es llamar a AsSequential para deshabilitar la paralelización.

Para todos los demás operadores de consulta, tienes la opción de utilizar particionamiento por rango o por fragmentos. De forma predeterminada:

- Si la secuencia de entrada es indexable (si es un arreglo o implementa IList<T>), PLINQ elige el particionamiento por rango.
- De lo contrario, PLINQ elige el particionamiento por fragmentos.

En resumen, el particionamiento por rango es más rápido con secuencias largas en las que cada elemento tarda una cantidad similar de tiempo de CPU en procesarse. De lo contrario, el particionamiento por fragmentos suele ser más rápido.

Para forzar el particionamiento por rango:

- Si la consulta comienza con Enumerable.Range, reemplaza ese método por ParallelEnumerable.Range.
- De lo contrario, simplemente llama a ToList o ToArray en la secuencia de entrada (obviamente, esto incurre en un costo de rendimiento en sí mismo, que debes tener en cuenta).



ParallelEnumerable.Range no es simplemente un atajo para llamar a Enumerable.Range(...).AsParallel(). Cambia el rendimiento de la consulta activando el particionamiento por rango.

Para forzar el particionamiento por fragmentos, envuelve la secuencia de entrada en una llamada a Partitioner.Create (en System.Collection.Concurrent), de la siguiente manera:

```
int[] numbers = { 3, 4, 5, 6, 7, 8, 9 };
var parallelQuery =
  Partitioner.Create (numbers, true).AsParallel()
  .Where (...)
```

El segundo argumento de Partitioner.Create indica que deseas equilibrar la carga de la consulta, lo cual es otra forma de decir que deseas utilizar el particionamiento por fragmentos.

El particionamiento por fragmentos funciona haciendo que cada hilo de trabajo tome periódicamente pequeños "fragmentos" de elementos de la secuencia de entrada para procesarlos (ver Figura 22-4). PLINQ comienza asignando fragmentos muy pequeños (uno o dos elementos a la vez). Luego, aumenta el tamaño del fragmento a medida que avanza la consulta: esto asegura que las secuencias pequeñas se paralelicen de manera efectiva y que las secuencias grandes no causen un exceso de ida y vuelta.

Si un hilo de trabajo obtiene elementos "fáciles" (que se procesan rápidamente), terminará obteniendo más fragmentos. Este sistema mantiene todos los hilos igualmente ocupados (y los núcleos "equilibrados"); la única desventaja es que obtener elementos de la secuencia de entrada compartida requiere sincronización (normalmente un bloqueo exclusivo), lo que puede generar cierta sobrecarga y contención.

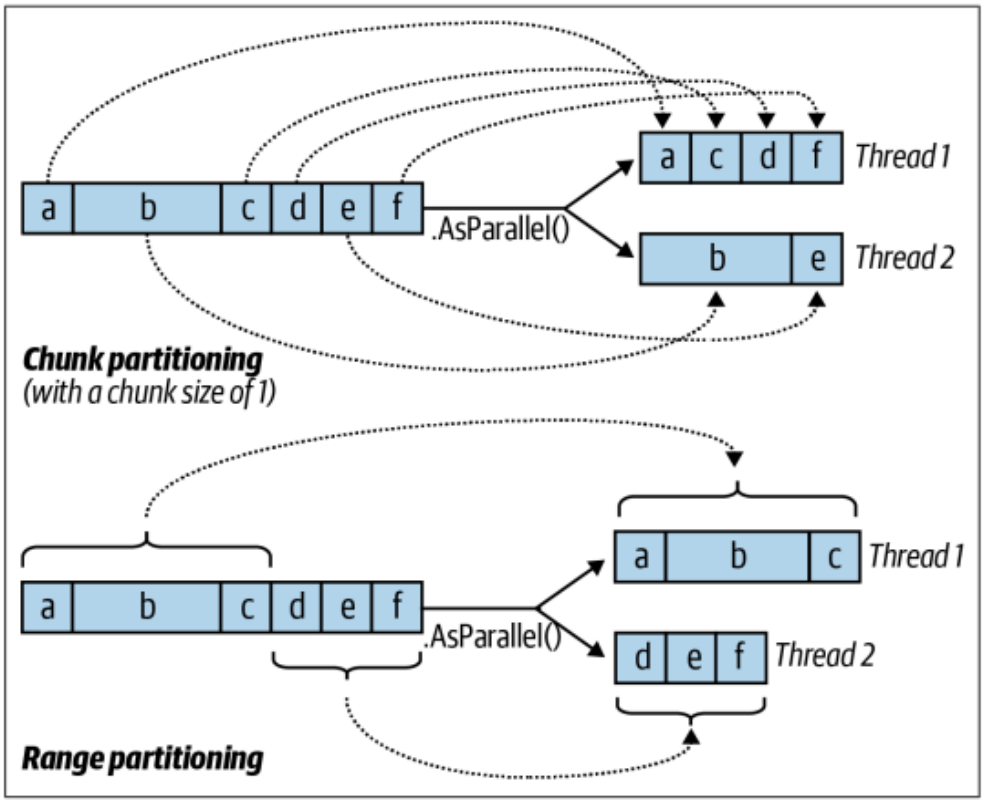


Figure 22-4. Chunk versus range partitioning

El particionamiento por rango omite la enumeración normal de la entrada y preasigna un número igual de elementos a cada hilo de trabajo, evitando la contención en la secuencia de entrada. Sin embargo, si algunos hilos obtienen elementos fáciles y terminan temprano, se quedan inactivos mientras los hilos restantes continúan trabajando. Nuestro ejemplo anterior del cálculo de números primos podría tener un rendimiento deficiente con el particionamiento por rango. Un ejemplo en el que el particionamiento por rango funcionaría bien es al calcular la suma de las raíces cuadradas de los primeros 10 millones de enteros:

```
ParallelEnumerable.Range (1, 10000000).Sum (i => Math.Sqrt (i))
```

ParallelEnumerable.Range devuelve un ParallelQuery<T>, por lo que no es necesario llamar posteriormente a AsParallel.



El particionamiento por rango no necesariamente asigna rangos de elementos en bloques contiguos, en su lugar puede elegir una estrategia de "rayado" (striping). Por ejemplo, si hay dos hilos de trabajo, un hilo podría procesar elementos impares mientras que el otro procesa elementos pares. El operador TakeWhile casi seguramente activará una estrategia de rayado para evitar procesar innecesariamente elementos posteriores en la secuencia.

Optimización de agregaciones personalizadas (Optimizing custom aggregations)

PLINQ paraleliza eficientemente los operadores Sum, Average, Min y Max sin intervención adicional. Sin embargo, el operador Aggregate presenta desafíos especiales para PLINQ. Como se describe en el Capítulo 9, Aggregate realiza agregaciones personalizadas. Por ejemplo, la siguiente línea de código suma una secuencia de números, imitando el operador Sum:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate (0, (total, n) => total + n);   // 6
```

También vimos en el Capítulo 9 que para agregaciones sin semilla, el delegado proporcionado debe ser asociativo y conmutativo. PLINQ dará resultados incorrectos si se viola esta regla, ya que extrae múltiples semillas de la secuencia de entrada para agregar simultáneamente varias particiones de la secuencia.

Las agregaciones explícitamente sembradas pueden parecer una opción segura con PLINQ, pero desafortunadamente, por lo general se ejecutan secuencialmente debido a la dependencia de una sola semilla. Para mitigar esto, PLINQ proporciona otra sobrecarga de Aggregate que te permite especificar múltiples semillas, o más bien, una función generadora de semillas. Para cada hilo, ejecuta esta función para generar una semilla separada, que se convierte en un acumulador local del hilo en el que se agregan elementos localmente.

También debes proporcionar una función para indicar cómo combinar los acumuladores locales y principales. Finalmente, esta sobrecarga de Aggregate (de manera un tanto gratuita) espera un delegado para realizar cualquier transformación final en el resultado (esto se puede lograr fácilmente ejecutando alguna función en el resultado por tu cuenta después). Por lo tanto, aquí están los cuatro delegados, en el orden en que se pasan:

seedFactory

: devuelve un nuevo acumulador local

updateAccumulatorFunc

: agrega un elemento a un acumulador local

combineAccumulatorFunc

: combina un acumulador local con el acumulador principal

resultSelector

: aplica cualquier transformación final al resultado final



En escenarios simples, puedes especificar un valor de semilla en lugar de una función generadora de semilla. Esta táctica falla cuando la semilla es un tipo de referencia que deseas modificar, ya que la misma instancia será compartida por cada hilo.

Aquí tienes un ejemplo muy simple que suma los valores en un array de números:

```
numbers.AsParallel().Aggregate (
 () => 0,                                        // seedFactory
  (localTotal, n) => localTotal + n,             // updateAccumulatorFunc

 (mainTot, localTot) => mainTot + localTot,     // combineAccumulatorFunc
 finalResult => finalResult)                    // resultSelector
```

Este ejemplo es artificial en el sentido de que podríamos obtener la misma respuesta de manera igualmente eficiente utilizando enfoques más simples (como una agregación no sembrada, o mejor aún, el operador Sum). Para dar un ejemplo más realista, supongamos que queremos calcular la frecuencia de cada letra en el alfabeto inglés en una cadena dada. Una solución secuencial simple podría verse así:

```
string text = "Let's suppose this is a really long string";
var letterFrequencies = new int[26];
foreach (char c in text)
{
  int index = char.ToUpper (c) - 'A';
  if (index >= 0 && index < 26) letterFrequencies [index]++;
};
```



Un ejemplo de cuándo el texto de entrada podría ser muy largo es en la secuenciación de genes. El "alfabeto" entonces consistiría en las letras a, c, g y t.

Para paralelizar esto, podríamos reemplazar la instrucción foreach con una llamada a Parallel.ForEach (que cubriremos en la siguiente sección), pero eso nos dejaría lidiar con problemas de concurrencia en el array compartido. Y el uso de bloqueos alrededor del acceso a ese array prácticamente eliminaría el potencial de paralelización.

Aggregate ofrece una solución ordenada. El acumulador, en este caso, es un array igual al array de frecuenciaLetras en nuestro ejemplo anterior. Aquí tienes una versión secuencial utilizando Aggregate:

```
int[] result =
  text.Aggregate (
    new int[26],                // Create the "accumulator"
    (letterFrequencies, c) =>   // Aggregate a letter into the accumulator
    {
      int index = char.ToUpper (c) - 'A';
      if (index >= 0 && index < 26) letterFrequencies [index]++;
      return letterFrequencies;
    });
```

Y ahora la versión paralela, utilizando la sobrecarga especial de PLINQ:

```
int[] result =
  text.AsParallel().Aggregate (
   () => new int[26],             // Create a new local accumulator

    (localFrequencies, c) =>       // Aggregate into the local accumulator
    {
      int index = char.ToUpper (c) - 'A';
      if (index >= 0 && index < 26) localFrequencies [index]++;
      return localFrequencies;
    },
                                   // Aggregate local->main accumulator
    (mainFreq, localFreq) =>


      mainFreq.Zip (localFreq, (f1, f2) => f1 + f2).ToArray(),

    finalResult => finalResult     // Perform any final transformation
  );                               // on the end result.
```

Observa que la función de acumulación local modifica el arreglo localFrequencies. Esta capacidad para realizar esta optimización es importante y legítima porque localFrequencies es local a cada hilo.

La clase Parallel

PFX proporciona una forma básica de paralelismo estructurado a través de tres métodos estáticos en la clase Parallel:

Parallel.Invoke

Ejecuta un conjunto de delegados en paralelo.

Parallel.For

Realiza el equivalente en paralelo de un bucle for en C#.

Parallel.ForEach

Realiza el equivalente en paralelo de un bucle foreach en C#.

Los tres métodos bloquean hasta que se completa todo el trabajo. Al igual que con PLINQ, después de una excepción no controlada, los trabajadores restantes se detienen después de su iteración actual y la excepción (o excepciones) se lanzan de vuelta al llamador, envuelta en una AggregateException (ver "Trabajando con AggregateException" en la página 940).

Parallel.Invoke

Parallel.Invoke ejecuta un conjunto de delegados Action en paralelo y luego espera a que se completen. La versión más simple del método se define de la siguiente manera:

```
public static void Invoke (params Action[] actions);
```

Al igual que con PLINQ, los métodos Parallel.* están optimizados para tareas de cómputo y no para tareas de E/S. Sin embargo, descargar dos páginas web al mismo tiempo proporciona una forma sencilla de demostrar Parallel.Invoke:

```
Parallel.Invoke (
 () => new WebClient().DownloadFile ("http://www.linqpad.net", "lp.html"),
 () => new WebClient().DownloadFile ("http://microsoft.com", "ms.html"));
```

A primera vista, esto parece ser un atajo conveniente para crear y esperar en dos objetos de tipo Task vinculados a hilos. Pero hay una diferencia importante: Parallel.Invoke sigue funcionando eficientemente si pasas un arreglo con un millón de delegados. Esto se debe a que divide grandes cantidades de elementos en lotes que asigna a un puñado de tareas subyacentes en lugar de crear una tarea separada para cada delegado.

Al igual que con todos los métodos de Parallel, eres responsable de recopilar los resultados. Esto significa que debes tener en cuenta la seguridad de hilos. Por ejemplo, lo siguiente no es seguro para hilos:

```
var data = new List<string>();
Parallel.Invoke (
 () => data.Add (new WebClient().DownloadString ("http://www.foo.com")),
 () => data.Add (new WebClient().DownloadString ("http://www.far.com")));
```

Bloquear alrededor de la adición a la lista resolvería este problema, aunque el bloqueo crearía un cuello de botella si tuvieras un arreglo mucho más grande de delegados que se ejecutan rápidamente. Una solución mejor es utilizar las colecciones seguras para subprocesos, que se cubren en secciones posteriores. ConcurrentBag sería ideal en este caso.

Parallel.Invoke también tiene una sobrecarga para aceptar un objeto ParallelOptions:

```
public static void Invoke (ParallelOptions options,
                           params Action[] actions);
```

Con ParallelOptions, puedes insertar un token de cancelación, limitar la concurrencia máxima y especificar un programador de tareas personalizado. Un token de cancelación es relevante cuando estás ejecutando (aproximadamente) más tareas de las que tienes núcleos: al cancelar, se abandonarán los delegados que no se hayan iniciado. Sin embargo, los delegados que ya se estén ejecutando continuarán hasta su finalización. Consulta "Cancelación" en la página 918 para ver un ejemplo de cómo usar tokens de cancelación.

Parallel.For y Parallel.ForEach

Parallel.For y Parallel.ForEach realizan el equivalente de un bucle for y foreach en C#, pero cada iteración se ejecuta en paralelo en lugar de secuencialmente. Aquí están sus firmas más simples:

```
public static ParallelLoopResult For (
  int fromInclusive, int toExclusive, Action<int> body)

public static ParallelLoopResult ForEach<TSource> (
  IEnumerable<TSource> source, Action<TSource> body)
```

Este bucle for secuencial:

```
for (int i = 0; i < 100; i++)
  Foo (i);
```

se paraleliza de la siguiente manera:

```
Parallel.For (0, 100, i => Foo (i));
```

o más simplemente:

```
Parallel.For (0, 100, Foo);
```

Y este foreach secuencial:

```
foreach (char c in "Hello, world")
  Foo (c);
```

se paraleliza de la siguiente manera.

```
Parallel.ForEach ("Hello, world", Foo);
```

Para dar un ejemplo práctico, si importamos el espacio de nombres System.Security.Cryptography, podemos generar seis pares de claves pública/privada en paralelo de la siguiente manera:

```
var keyPairs = new string[6];

Parallel.For (0, keyPairs.Length,
              i => keyPairs[i] = RSA.Create().ToXmlString (true));
```

Al igual que con Parallel.Invoke, podemos proporcionar a Parallel.For y Parallel.ForEach una gran cantidad de elementos de trabajo y serán asignados eficientemente a unas pocas tareas.



La última consulta también se puede realizar con PLINQ:

```
string[] keyPairs =
  ParallelEnumerable.Range (0, 6)
  .Select (i => RSA.Create().ToXmlString (true))
  .ToArray();
```

Bucles externos frente a internos

Parallel.For y Parallel.ForEach suelen funcionar mejor en bucles externos en lugar de bucles internos. Esto se debe a que con el primero, estás ofreciendo fragmentos más grandes de trabajo para paralelizar, lo que diluye la sobrecarga de gestión. Paralelizar tanto los bucles internos como externos suele ser innecesario. En el siguiente ejemplo, normalmente necesitaríamos más de 100 núcleos para beneficiarnos de la paralelización interna:

```
Parallel.For (0, 100, i =>
{
  Parallel.For (0, 50, j => Foo (i, j));   // Sequential would be better
});                                        // for the inner loop.
```

ForEach indexado

A veces es útil conocer el índice de iteración del bucle. Con un foreach secuencial, es fácil:

```
int i = 0;
foreach (char c in "Hello, world")
  Console.WriteLine (c.ToString() + i++);
```

Sin embargo, incrementar una variable compartida no es seguro en un contexto paralelo. En su lugar, debes utilizar la siguiente versión de ForEach:

```
public static ParallelLoopResult ForEach<TSource> (
  IEnumerable<TSource> source, Action<TSource,ParallelLoopState,long> body)
```

Ignoraremos ParallelLoopState (que se cubre en la siguiente sección). Por ahora, nos interesa el tercer parámetro de tipo long de la acción, que indica el índice del bucle:

```
Parallel.ForEach ("Hello, world", (c, state, i) =>
{
   Console.WriteLine (c.ToString() + i);
});
```

Para poner esto en un contexto práctico, revisemos el corrector ortográfico que escribimos con PLINQ. El siguiente código carga un diccionario junto con un array de un millón de palabras para probar:

```
if (!File.Exists ("WordLookup.txt"))    // Contains about 150,000 words
  new WebClient().DownloadFile (
    "http://www.albahari.com/ispell/allwords.txt", "WordLookup.txt");

var wordLookup = new HashSet<string> (
  File.ReadAllLines ("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);

var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();

wordsToTest [12345] = "woozsh";     // Introduce a couple
wordsToTest [23456] = "wubsie";     // of spelling mistakes.
```

Podemos realizar la corrección ortográfica en nuestro array wordsToTest utilizando la versión indexada de Parallel.ForEach, de la siguiente manera:

```
var misspellings = new ConcurrentBag<Tuple<int,string>>();

Parallel.ForEach (wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains (word))
    misspellings.Add (Tuple.Create ((int) i, word));
});
```

Observa que tuvimos que recopilar los resultados en una colección segura para subprocesos. Esta es la desventaja en comparación con el uso de PLINQ. La ventaja sobre PLINQ es que evitamos el costo de aplicar un operador de consulta Select indexado, que es menos eficiente que un ForEach indexado.

ParallelLoopState: Rompiendo bucles tempranamente

Debido a que el cuerpo del bucle en un Parallel.For o Parallel.ForEach es un delegado, no puedes salir del bucle tempranamente con una instrucción break. En su lugar, debes llamar a Break o Stop en un objeto ParallelLoopState:

```
public class ParallelLoopState
{
  public void Break();
  public void Stop();

  public bool IsExceptional { get; }
  public bool IsStopped { get; }
  public long? LowestBreakIteration { get; }
  public bool ShouldExitCurrentIteration { get; }
}
```

Obtener un `ParallelLoopState` es fácil: todas las versiones de `For` y `ForEach` están sobrecargadas para aceptar cuerpos de bucle del tipo `Action<TSource, ParallelLoopState>`. Entonces, para paralelizar esto:

```
foreach (char c in "Hello, world")
  if (c == ',')
    break;
  else
    Console.Write (c);
```

haz esto:

```
Parallel.ForEach ("Hello, world", (c, loopState) =>
{
  if (c == ',')
    loopState.Break();
  else
    Console.Write (c);
});

// OUTPUT: Hlloe
```

Puedes ver en la salida que los cuerpos del bucle pueden completarse en un orden aleatorio. Aparte de esta diferencia, llamar a `Break` produce al menos los mismos elementos que ejecutar

el bucle secuencialmente: este ejemplo siempre mostrará al menos las letras H, e, l, l y o en algún orden. En contraste, llamar a `Stop` en lugar de `Break` obliga a todos los hilos a finalizar inmediatamente después de su iteración actual. En nuestro ejemplo, llamar a `Stop` podría darnos un subconjunto de las letras H, e, l, l y o si otro hilo se retrasara.

Llamar a `Stop` es útil cuando has encontrado algo que estabas buscando, o cuando algo ha salido mal y no necesitas ver los resultados.



Los métodos `Parallel.For` y `Parallel.ForEach` devuelven un objeto `ParallelLoopResult` que expone propiedades llamadas `IsCompleted` y `LowestBreakIteration`. Estas propiedades te indican si el bucle se ejecutó hasta su finalización; si no lo hizo, indican en qué ciclo se interrumpió el bucle.

Si `LowestBreakIteration` devuelve `null`, significa que llamaste a `Stop` (en lugar de `Break`) en el bucle.

Si el cuerpo del bucle es largo, es posible que desees que otros hilos se detengan a mitad del cuerpo del método en caso de una interrupción (`Break`) o una detención (`Stop`) temprana. Puedes hacer esto consultando la propiedad `ShouldExitCurrentIteration` en varios lugares de tu código; esta propiedad se vuelve `true` inmediatamente después de un `Stop` o poco después de un `Break`.



`ShouldExitCurrentIteration` también se vuelve `true` después de una solicitud de cancelación, o si se produce una excepción en el bucle.

`IsExceptional` te permite saber si se ha producido una excepción en otro hilo. Cualquier excepción no controlada hará que el bucle se detenga después de la iteración actual de cada hilo: para evitar esto, debes manejar explícitamente las excepciones en tu código.

Optimización con valores locales

Parallel.For y Parallel.ForEach ofrecen una serie de sobrecargas que incluyen un argumento de tipo genérico llamado TLocal. Estas sobrecargas están diseñadas para ayudar a optimizar la recopilación de datos en bucles intensivos en iteraciones. La más simple es la siguiente:

```
public static ParallelLoopResult For <TLocal> (
  int fromInclusive,
  int toExclusive,
  Func <TLocal> localInit,
  Func <int, ParallelLoopState, TLocal, TLocal> body,
  Action <TLocal> localFinally);
```

Estos métodos rara vez son necesarios en la práctica, ya que los escenarios a los que apuntan están cubiertos principalmente por PLINQ (¡afortunadamente, porque estas sobrecargas pueden resultar un tanto intimidantes!).

Básicamente, el problema es el siguiente: supongamos que queremos sumar las raíces cuadradas de los números del 1 al 10,000,000. Calcular las raíces cuadradas de 10 millones de números se puede paralelizar fácilmente, pero sumar sus valores es problemático porque debemos bloquear la actualización del total:

```
object locker = new object();
double total = 0;
Parallel.For (1, 10000000,
              i => { lock (locker) total += Math.Sqrt (i); });
```

El beneficio de la paralelización se ve compensado por el costo de obtener 10 millones de bloqueos, además del bloqueo resultante.

Sin embargo, en realidad no necesitamos 10 millones de bloqueos. Imagina un equipo de voluntarios recogiendo una gran cantidad de basura. Si todos los trabajadores compartieran un solo contenedor de basura, los desplazamientos y la contención harían que el proceso fuera extremadamente ineficiente. La solución obvia es que cada trabajador tenga su propio contenedor de basura local, que se vacía ocasionalmente en el contenedor principal.

Las versiones de TLocal de For y ForEach funcionan exactamente de esta manera. Los voluntarios son los hilos internos del trabajo y el valor local representa un contenedor de basura local. Para que Parallel realice este trabajo, debes proporcionarle dos delegados adicionales que indiquen lo siguiente:

1. Cómo inicializar un nuevo valor local.

2. Cómo combinar una agregación local con el valor maestro.

Además, en lugar de que el delegado body devuelva void, debe devolver la nueva agregación para el valor local. Aquí tienes un ejemplo refactorizado:

```
object locker = new object();
double grandTotal = 0;

Parallel.For (1, 10000000,

  () => 0.0,                        // Initialize the local value.


   (i, state, localTotal) =>         // Body delegate. Notice that it
      localTotal + Math.Sqrt (i),    // returns the new local total.

   localTotal =>                                    // Add the local value
     { lock (locker) grandTotal += localTotal; }    // to the master value.
);
```

Aún debemos usar un bloqueo, pero solo alrededor de la agregación del valor local al gran total. Esto hace que el proceso sea considerablemente más eficiente.



Como se mencionó anteriormente, PLINQ a menudo es una buena opción en estos escenarios. Nuestro ejemplo se podría paralelizar con PLINQ de la siguiente manera:

```
ParallelEnumerable.Range (1, 10000000)
                  .Sum (i => Math.Sqrt (i))
```

(Observa que usamos ParallelEnumerable para forzar la partición del rango: esto mejora el rendimiento en este caso porque todos los números tomarán igual tiempo en procesar).

En escenarios más complejos, es posible que uses el operador Aggregate de LINQ en lugar de Sum. Si proporcionaras una función de semilla local, la situación sería algo análoga a proporcionar una función de valor local con Parallel.For.

Paralelismo de tareas

El paralelismo de tareas es el enfoque de nivel más bajo para la paralelización con PFX. Las clases para trabajar en este nivel están definidas en el espacio de nombres System.Threading.Tasks y comprenden lo siguiente:

Class	Purpose
`Task`	For managing a unit for work
`Task<TResult>`	For managing a unit for work with a return value
`TaskFactory`	For creating tasks
`TaskFactory<TResult>`	For creating tasks and continuations with the same return type
`TaskScheduler`	For managing the scheduling of tasks
`TaskCompletionSource`	For manually controlling a task's workflow

Hemos cubierto los conceptos básicos de las tareas en el Capítulo 14; en esta sección, exploramos características avanzadas de las tareas que están orientadas a la programación paralela:

- Ajuste de la programación de tareas.
- Establecimiento de una relación padre/hijo cuando una tarea se inicia desde otra.
- Uso avanzado de continuaciones.
- TaskFactory.



La biblioteca de Tareas Paralelas te permite crear cientos (o incluso miles) de tareas con un mínimo sobrecarga. Sin embargo, si deseas crear millones de tareas, deberás particionar esas tareas en unidades de trabajo más grandes para mantener la eficiencia. La clase Parallel y PLINQ hacen esto automáticamente.

 Visual Studio proporciona una ventana para monitorear tareas (Depurar → Ventana → Tareas Paralelas). Esto es equivalente a la ventana de Hilos, pero para tareas. La ventana de Parallel Stacks también tiene un modo especial para tareas.

Creación e inicio de tareas

Como se describe en el Capítulo 14, Task.Run crea e inicia una Task o Task<TResult>. Este método es en realidad un atajo para llamar a Task.Factory.StartNew, que permite una mayor flexibilidad a través de sobrecargas adicionales.

Especificar un objeto de estado

Task.Factory.StartNew te permite especificar un objeto de estado que se pasa al método objetivo. La firma del método objetivo debe entonces incluir un parámetro de tipo objeto:

```
var task = Task.Factory.StartNew (Greet, "Hello");
task.Wait();  // Wait for task to complete.

void Greet (object state) { Console.Write (state); }   // Hello
```

Esto evita el costo de la clausura requerida para ejecutar una expresión lambda que llama a Greet. Esta es una micro-optimización y rara vez es necesaria en la práctica, por lo que podemos usar el objeto de estado de manera más útil, que es asignar un nombre significativo a la tarea. Luego podemos usar la propiedad AsyncState para consultar su nombre:

```
var task = Task.Factory.StartNew (state => Greet ("Hello"), "Greeting");
Console.WriteLine (task.AsyncState);   // Greeting
task.Wait();

void Greet (string message) { Console.Write (message); }
```

 Visual Studio muestra el estado AsyncState de cada tarea en la ventana Parallel Tasks, por lo que tener un nombre significativo aquí puede facilitar considerablemente la depuración.

TaskCreationOptions

Puedes ajustar la ejecución de una tarea especificando un enum TaskCreationOptions al llamar a StartNew (o al instanciar una tarea). TaskCreationOptions es un enum de banderas con los siguientes valores (combinables):

```
LongRunning, PreferFairness, AttachedToParent
```

LongRunning sugiere al planificador que dedique un hilo a la tarea. Como describimos en el Capítulo 14, esto es beneficioso para tareas orientadas a E/S y para tareas de larga duración que

de otra manera podrían hacer que las tareas de corta duración esperen un tiempo irrazonable antes de ser programadas.

PreferFairness instruye al planificador a intentar asegurar que las tareas se programen en el orden en que se iniciaron. Normalmente, el planificador podría hacer lo contrario debido a que optimiza internamente la programación de tareas utilizando colas locales de robo de trabajo, una optimización que permite la creación de tareas secundarias sin incurrir en la sobrecarga de contención que de otro modo surgiría con una única cola de trabajo. Una tarea secundaria se crea especificando AttachedToParent.

Tareas secundarias (Child tasks)

Cuando una tarea inicia otra, puedes establecer opcionalmente una relación padre-hijo:

```
Task parent = Task.Factory.StartNew (() =>
{
  Console.WriteLine ("I am a parent");

  Task.Factory.StartNew (() =>        // Detached task
  {
    Console.WriteLine ("I am detached");
  });

  Task.Factory.StartNew (() =>        // Child task
  {
    Console.WriteLine ("I am a child");
  }, TaskCreationOptions.AttachedToParent);
});
```

Una tarea secundaria es especial en el sentido de que cuando esperas a que la tarea padre se complete, también esperas a que se completen todas las tareas secundarias. En este punto, cualquier excepción en las tareas secundarias se propagará hacia arriba:

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
var parent = Task.Factory.StartNew (() =>
{
  Task.Factory.StartNew (() =>   // Child
  {
    Task.Factory.StartNew (() => { throw null; }, atp);   // Grandchild
  }, atp);
});

// The following call throws a NullReferenceException (wrapped
// in nested AggregateExceptions):
parent.Wait();
```

Esto puede ser especialmente útil cuando una tarea secundaria es una continuación, como veremos pronto.

Esperar por varias tareas

Vimos en el Capítulo 14 que puedes esperar a una sola tarea llamando a su método Wait o accediendo a su propiedad Result (si es una Task<TResult>). También puedes esperar varias tareas a la vez utilizando los métodos estáticos Task.WaitAll (espera a que todas las tareas especificadas terminen) y Task.WaitAny (espera solo a que una tarea termine).

WaitAll es similar a esperar una tras otra cada tarea, pero es más eficiente en el sentido de que requiere (como máximo) solo un cambio de contexto. Además, si una o más tareas lanzan una excepción no controlada, WaitAll sigue esperando a todas las tareas y luego lanza una AggregateException que acumula las excepciones de cada tarea fallida (aquí es donde AggregateException es realmente útil). Es equivalente a hacer esto:

```
// Assume t1, t2 and t3 are tasks:
var exceptions = new List<Exception>();
try { t1.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
try { t2.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
try { t3.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
if (exceptions.Count > 0) throw new AggregateException (exceptions);
```

Llamar a WaitAny es equivalente a esperar en un ManualResetEventSlim que se señaliza por cada tarea a medida que se completa.

Además de un tiempo de espera, también puedes pasar un token de cancelación a los métodos Wait. Esto te permite cancelar la espera, pero no la tarea en sí misma.

Canceling Tasks (Cancelando tareas)

Puedes pasar opcionalmente un token de cancelación al iniciar una tarea. Luego, si ocurre una cancelación a través de ese token, la tarea misma entra en el estado "Canceled" (cancelado):

```
var cts = new CancellationTokenSource();
CancellationToken token = cts.Token;
cts.CancelAfter (500);

Task task = Task.Factory.StartNew (() =>
{
  Thread.Sleep (1000);
  token.ThrowIfCancellationRequested();  // Check for cancellation request
}, token);

try { task.Wait(); }
catch (AggregateException ex)
{
  Console.WriteLine (ex.InnerException is TaskCanceledException);  // True
  Console.WriteLine (task.IsCanceled);                             // True
  Console.WriteLine (task.Status);                              // Canceled
}
```

TaskCanceledException es una subclase de OperationCanceledException. Si deseas lanzar explícitamente una OperationCanceledException (en lugar de llamar a token.ThrowIfCancellationRequested), debes pasar el token de cancelación al constructor de OperationCanceledException. Si no lo haces, la tarea no tendrá el estado TaskStatus.Canceled y no activará las continuaciones configuradas con OnlyOnCanceled.

Si una tarea se cancela antes de que se inicie, no se programará y en su lugar se lanzará inmediatamente una OperationCanceledException en la tarea.

Debido a que los tokens de cancelación son reconocidos por otras API, puedes pasarlos a otras construcciones y las cancelaciones se propagarán sin problemas:

```
var cancelSource = new CancellationTokenSource();
CancellationToken token = cancelSource.Token;

Task task = Task.Factory.StartNew (() =>
{
  // Pass our cancellation token into a PLINQ query:
  var query = someSequence.AsParallel().WithCancellation (token)...
  ... enumerate query ...
});
```

Llamar a Cancel en cancelSource en este ejemplo cancelará la consulta PLINQ, lo cual lanzará una OperationCanceledException en el cuerpo de la tarea y luego cancelará la tarea.



Los tokens de cancelación que puedes pasar a métodos como Wait y CancelAndWait te permiten cancelar la operación de espera y no la tarea en sí misma.

Continuaciones

El método ContinueWith ejecuta un delegado inmediatamente después de que una tarea finaliza:

```
Task task1 = Task.Factory.StartNew (() => Console.Write ("antecedant.."));
Task task2 = task1.ContinueWith (ant => Console.Write ("..continuation"));
```

Tan pronto como la tarea task1 (el antecedente) se completa, falla o se cancela, la tarea task2 (la continuación) comienza. (Si task1 se hubiera completado antes de que se ejecutara la segunda línea de código, task2 se programaría para ejecutarse de inmediato). El argumento ant pasado a la expresión lambda de la continuación es una referencia a la tarea antecedente. ContinueWith en sí mismo devuelve una tarea, lo que facilita agregar más continuaciones.

De forma predeterminada, las tareas antecedentes y de continuación pueden ejecutarse en hilos diferentes. Puedes forzar que se ejecuten en el mismo hilo especificando TaskContinuationOptions.ExecuteSynchronously al llamar a ContinueWith: esto puede mejorar el rendimiento en continuaciones muy finas al reducir la indirección.

Continuaciones y Task<TResult>

Al igual que las tareas ordinarias, las continuaciones pueden ser del tipo Task<TResult> y devolver datos. En el siguiente ejemplo, calculamos Math.Sqrt(8*2) usando una serie de tareas encadenadas y luego escribimos el resultado:

```
Task.Factory.StartNew<int> (() => 8)
  .ContinueWith (ant => ant.Result * 2)
  .ContinueWith (ant => Math.Sqrt (ant.Result))
  .ContinueWith (ant => Console.WriteLine (ant.Result));   // 4
```

Nuestro ejemplo es algo ficticio para simplificar; en la vida real, estas expresiones lambda llamarían a funciones intensivas en cómputo.

Continuaciones y excepciones

Una continuación puede saber si un antecedente falló consultando la propiedad Exception de la tarea antecedente, o simplemente invocando Result / Wait y capturando la AggregateException resultante. Si un antecedente falla y la continuación no hace ninguna de estas cosas, se considera que la excepción no se ha observado y se dispara el evento estático TaskScheduler.UnobservedTaskException cuando la tarea se recolecta más tarde.

Un patrón seguro es relanzar las excepciones de los antecedentes. Mientras se espere en la continuación, la excepción se propagará y se relanzará hacia el que espera:

```
Task continuation = Task.Factory.StartNew     (()  => { throw null; })
                                .ContinueWith (ant =>
  {
    ant.Wait();
    // Continue processing...
  });

continuation.Wait();    // Exception is now thrown back to caller.
```

Otra forma de manejar las excepciones es especificar continuaciones diferentes para resultados excepcionales y no excepcionales. Esto se hace con TaskContinuationOptions:

```
Task task1 = Task.Factory.StartNew (() => { throw null; });

Task error = task1.ContinueWith (ant => Console.Write (ant.Exception),
                                 TaskContinuationOptions.OnlyOnFaulted);

Task ok = task1.ContinueWith (ant => Console.Write ("Success!"),
                              TaskContinuationOptions.NotOnFaulted);
```

Este patrón es particularmente útil en conjunto con tareas secundarias (child tasks), como verás muy pronto.

El siguiente método de extensión "ignora" las excepciones no manejadas de una tarea:

```
public static void IgnoreExceptions (this Task task)
{
  task.ContinueWith (t => { var ignore = t.Exception; },
    TaskContinuationOptions.OnlyOnFaulted);
}
```

(Esto podría mejorarse agregando código para registrar la excepción). Así es como se usaría:

```
Task.Factory.StartNew (() => { throw null; }).IgnoreExceptions();
```

Continuaciones y tareas secundarias (child tasks)

Una característica poderosa de las continuaciones es que se inician solo cuando todas las tareas secundarias (child tasks) han finalizado (ver Figura 22-5). En ese momento, cualquier excepción lanzada por las tareas secundarias se envía a la continuación.

En el siguiente ejemplo, iniciamos tres tareas secundarias, cada una de las cuales lanza una NullReferenceException. Luego, capturamos todas ellas de una vez a través de una continuación en la tarea padre:

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
Task.Factory.StartNew (() =>
{
  Task.Factory.StartNew (() => { throw null; }, atp);
  Task.Factory.StartNew (() => { throw null; }, atp);
  Task.Factory.StartNew (() => { throw null; }, atp);
})
.ContinueWith (p => Console.WriteLine (p.Exception),
                    TaskContinuationOptions.OnlyOnFaulted);
```

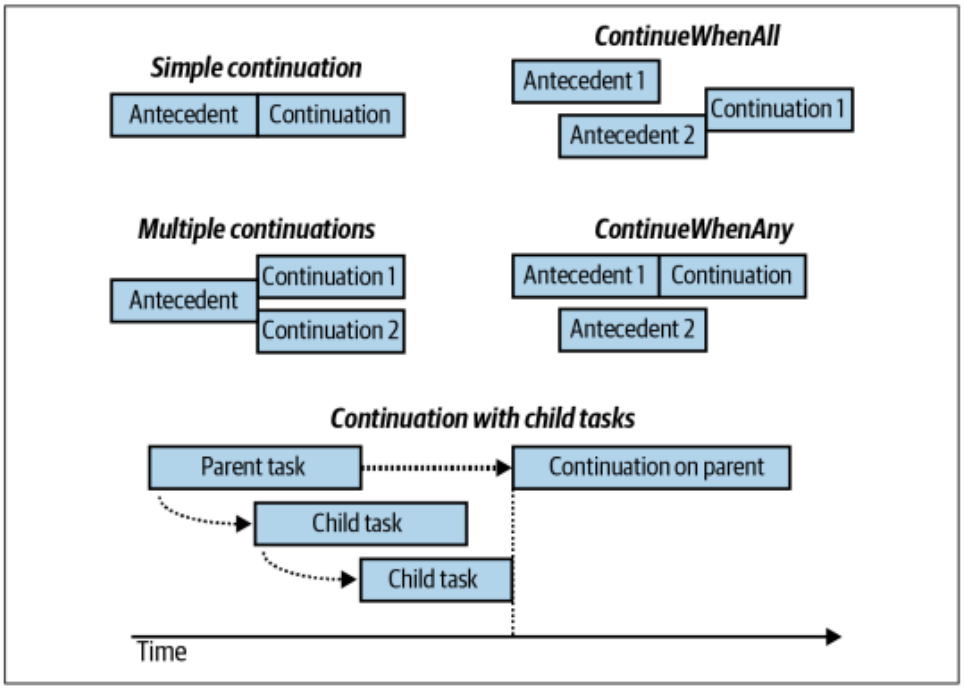


Figure 22-5. Continuations

Continuaciones condicionales

Por defecto, una continuación se programa incondicionalmente, ya sea que el antecedente se complete, lance una excepción o sea cancelado. Puedes modificar este comportamiento utilizando un conjunto de indicadores (flags) incluidos en el enum TaskContinuationOptions. A continuación se presentan los tres indicadores principales que controlan la continuación condicional:

```
NotOnRanToCompletion = 0x10000,
NotOnFaulted = 0x20000,
NotOnCanceled = 0x40000,
```

Estas banderas son restadoras en el sentido de que cuanto más se apliquen, menos probable es que se ejecute la continuación. Para mayor comodidad, también existen los siguientes valores predefinidos combinados:

```
OnlyOnRanToCompletion = NotOnFaulted | NotOnCanceled,
OnlyOnFaulted = NotOnRanToCompletion | NotOnCanceled,
OnlyOnCanceled = NotOnRanToCompletion | NotOnFaulted
```

(Combinar todas las banderas Not* [NotOnRanToCompletion, NotOnFaulted, NotOnCanceled] no tiene sentido, ya que resultaría en que la continuación siempre se cancela).

"RanToCompletion" significa que el antecedente se completó sin cancelación ni excepciones no controladas.

"Faulted" significa que se lanzó una excepción no controlada en el antecedente.

"Canceled" significa una de dos cosas:

- El antecedente fue cancelado a través de su token de cancelación. En otras palabras, se lanzó una OperationCanceledException en el antecedente, cuya propiedad CancellationToken coincidió con la que se pasó al antecedente cuando se inició.
- El antecedente fue cancelado implícitamente porque no cumplió con un predicado de continuación condicional.

Es esencial comprender que cuando una continuación no se ejecuta debido a estas banderas, la continuación no se olvida ni se abandona, se cancela. Esto significa que cualquier continuación en la continuación en sí se ejecutará a menos que se predique con NotOnCanceled. Por ejemplo, considera lo siguiente:

```
Task t1 = Task.Factory.StartNew (...);

Task fault = t1.ContinueWith (ant => Console.WriteLine ("fault"),
                              TaskContinuationOptions.OnlyOnFaulted);

Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"));
```

Como está, t3 siempre se programará, incluso si t1 no lanza una excepción (ver Figura 22-6). Esto se debe a que si t1 tiene éxito, la tarea fault se cancelará y, sin restricciones de continuación en t3, t3 se ejecutará incondicionalmente.

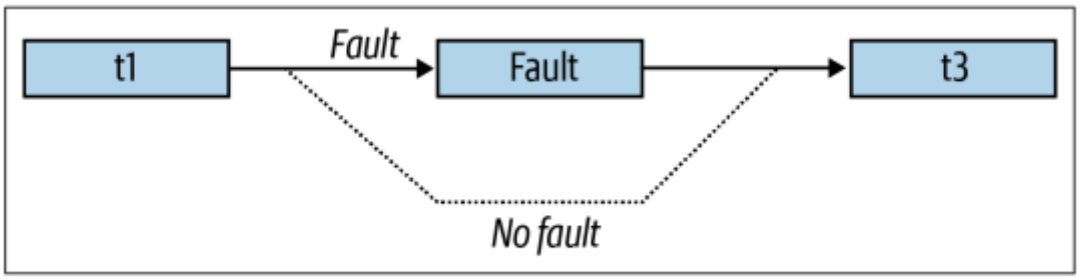


Figure 22-6. Conditional continuations

Si queremos que t3 se ejecute solo si fault se ejecuta realmente, debemos hacer lo siguiente:

```
Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"),
                              TaskContinuationOptions.NotOnCanceled);
```

(Alternativamente, podríamos especificar OnlyOnRanToCompletion; la diferencia es que t3 no se ejecutaría si se produce una excepción dentro de fault).

Continuaciones con múltiples antecedentes

Puedes programar una continuación para que se ejecute en función de la finalización de múltiples antecedentes con los métodos ContinueWhenAll y ContinueWhenAny de la clase TaskFactory. Sin embargo, estos métodos se han vuelto redundantes con la introducción de los combinadores de tareas discutidos en el Capítulo 14 (WhenAll y WhenAny).

Específicamente, dadas las siguientes tareas:

```
var task1 = Task.Run (() => Console.Write ("X"));
var task2 = Task.Run (() => Console.Write ("Y"));
```

Podemos programar una continuación para que se ejecute cuando ambas tareas se completen de la siguiente manera:

```
var continuation = Task.Factory.ContinueWhenAll (
  new[] { task1, task2 }, tasks => Console.WriteLine ("Done"));
```

Aquí está el mismo resultado con el combinador de tareas WhenAll:

```
var continuation = Task.WhenAll (task1, task2)
                       .ContinueWith (ant => Console.WriteLine ("Done"));
```

Múltiples continuaciones en un solo antecedente

Llamar a ContinueWith más de una vez en la misma tarea crea múltiples continuaciones en un solo antecedente. Cuando el antecedente finaliza, todas las continuaciones comenzarán juntas (a menos que especifiques TaskContinuationOptions.ExecuteSynchronously, en cuyo caso las continuaciones se ejecutarán secuencialmente).

El siguiente código espera un segundo y luego escribe XY o YX:

```
var t = Task.Factory.StartNew (() => Thread.Sleep (1000));
t.ContinueWith (ant => Console.Write ("X"));
t.ContinueWith (ant => Console.Write ("Y"));
```

Planificadores de tareas (Task Schedulers)

Un planificador de tareas asigna tareas a hilos y está representado por la clase abstracta TaskScheduler. .NET proporciona dos implementaciones concretas: el planificador predeterminado que funciona junto con el grupo de hilos de CLR (Common Language Runtime) y el planificador de contexto de sincronización. Este último está diseñado principalmente para ayudar con el modelo de subprocesos de WPF (Windows Presentation Foundation) y Windows Forms, que requiere que los elementos y controles de la interfaz de usuario sean accedidos solo

desde el hilo que los creó (ver "Subprocesos en aplicaciones de cliente rico" en la página 621). Al capturarlo, podemos indicar a una tarea o una continuación que se ejecute en este contexto:

```
// Suppose we are on a UI thread in a Windows Forms / WPF application:
_uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
```

Suponiendo que "Foo" es un método de cómputo que devuelve una cadena y "lblResult" es una etiqueta de WPF o Windows Forms, podríamos actualizar de manera segura la etiqueta después de que la operación se complete de la siguiente manera:

```
Task.Run (() => Foo())
  .ContinueWith (ant => lblResult.Content = ant.Result, _uiScheduler);
```

Por supuesto, en este tipo de casos, sería más común utilizar las funciones asincrónicas de C#.

También es posible escribir nuestro propio planificador de tareas (mediante la creación de una subclase de TaskScheduler), aunque esto se hace solo en escenarios muy especializados. Para una programación personalizada, lo más común es utilizar TaskCompletionSource.

TaskFactory

Cuando llamas a Task.Factory, estás llamando a una propiedad estática en Task que devuelve un objeto TaskFactory predeterminado. El propósito de una fábrica de tareas es crear tareas, específicamente tres tipos de tareas:

- Tareas "ordinarias" (a través de StartNew)
- Continuaciones con múltiples antecedentes (a través de ContinueWhenAll y ContinueWhenAny)
- Tareas que envuelven métodos que siguen los patrones obsoletos de APM (a través de FromAsync; ver "Patrones obsoletos" en la página 667)

Otra forma de crear tareas es instanciar Task y llamar a Start. Sin embargo, esto solo te permite crear tareas "ordinarias", no continuaciones.

Creando tus propias fábricas de tareas

TaskFactory no es una fábrica abstracta: en realidad puedes instanciar la clase, y esto es útil cuando deseas crear tareas repetidamente usando los mismos valores (no estándar) para TaskCreationOptions, TaskContinuationOptions o TaskScheduler. Por ejemplo, si queremos crear tareas principales de larga duración repetidamente, podríamos crear una fábrica personalizada de la siguiente manera:

```
var factory = new TaskFactory (
  TaskCreationOptions.LongRunning | TaskCreationOptions.AttachedToParent,
  TaskContinuationOptions.None);
```

Luego, simplemente debes llamar a StartNew en la fábrica para crear tareas:

```
Task task1 = factory.StartNew (Method1);
Task task2 = factory.StartNew (Method2);
...
```

Las opciones personalizadas de continuación se aplican al llamar a ContinueWhenAll y ContinueWhenAny.

Trabajando con AggregateException

Como hemos visto, PLINQ, la clase Parallel y las tareas automáticamente capturan las excepciones para el consumidor. Para entender por qué esto es esencial, considera la siguiente consulta LINQ, que arroja una DivideByZeroException en la primera iteración:

```
try
{
  var query = from i in Enumerable.Range (0, 1000000)
              select 100 / i;
  ...
}
catch (DivideByZeroException)
{
  ...
}
```

Si le pedimos a PLINQ que paralelice esta consulta y no se encarga de manejar las excepciones, es probable que se arroje una DivideByZeroException en un hilo separado, evitando nuestro bloque catch y causando que la aplicación termine abruptamente.

Por lo tanto, las excepciones se capturan automáticamente y se vuelven a lanzar al llamante. Pero desafortunadamente, no es tan simple como capturar una DivideByZeroException. Debido a que estas bibliotecas utilizan muchos hilos, es posible que se lancen dos o más excepciones simultáneamente. Para asegurarnos de que se informen todas las excepciones, las excepciones se envuelven en un contenedor AggregateException, que expone una propiedad InnerExceptions que contiene cada una de las excepciones capturadas:

```
try
{
  var query = from i in ParallelEnumerable.Range (0, 1000000)
              select 100 / i;
  // Enumerate query
  ...
}
catch (AggregateException aex)
{
  foreach (Exception ex in aex.InnerExceptions)
    Console.WriteLine (ex.Message);
}
```



Tanto PLINQ como la clase Parallel finalizan la ejecución de la consulta o del bucle al encontrar la primera excepción, sin procesar ningún otro elemento o cuerpo del bucle. Sin embargo, puede que se lancen más excepciones antes de que se complete el ciclo actual. La primera excepción en AggregateException es visible en la propiedad InnerException.

Flatten y Handle

La clase AggregateException proporciona un par de métodos para simplificar el manejo de excepciones: Flatten y Handle.

Flatten

Las AggregateExceptions a menudo contienen otras AggregateExceptions. Un ejemplo de cuándo esto puede suceder es si una tarea secundaria lanza una excepción. Puedes eliminar cualquier nivel de anidamiento para simplificar el manejo llamando a Flatten. Este método devuelve una nueva AggregateException con una lista plana de excepciones internas:

```
catch (AggregateException aex)
{
  foreach (Exception ex in aex.Flatten().InnerExceptions)
    myLogWriter.LogException (ex);
}
```

Handle

A veces, es útil capturar solo tipos específicos de excepciones y hacer que otros tipos se vuelvan a lanzar. El método Handle en AggregateException proporciona una forma más rápida de hacer esto. Acepta un predicado de excepción que se ejecuta en cada excepción interna:

```
public void Handle (Func<Exception, bool> predicate)
```

Si el predicado devuelve true, considera que esa excepción está "manejada". Después de que se haya ejecutado el delegado en cada excepción, ocurre lo siguiente:

- Si todas las excepciones fueron "manejadas" (el delegado devolvió true), la excepción no se vuelve a lanzar.
- Si hubo alguna excepción para la cual el delegado devolvió false ("no manejada"), se construye una nueva AggregateException que contiene esas excepciones y se vuelve a lanzar.

Por ejemplo, lo siguiente termina volviendo a lanzar otra AggregateException que contiene una única NullReferenceException:

```
var parent = Task.Factory.StartNew (() =>
{
  // We'll throw 3 exceptions at once using 3 child tasks:

  int[] numbers = { 0 };

  var childFactory = new TaskFactory
   (TaskCreationOptions.AttachedToParent, TaskContinuationOptions.None);

  childFactory.StartNew (() => 5 / numbers[0]);   // Division by zero
  childFactory.StartNew (() => numbers [1]);      // Index out of range
  childFactory.StartNew (() => { throw null; });  // Null reference
});

try { parent.Wait(); }
catch (AggregateException aex)
{
  aex.Flatten().Handle (ex =>   // Note that we still need to call Flatten
  {
    if (ex is DivideByZeroException)
    {
      Console.WriteLine ("Divide by zero");
      return true;                          // This exception is "handled"
    }
    if (ex is IndexOutOfRangeException)
    {
      Console.WriteLine ("Index out of range");
      return true;                          // This exception is "handled"
    }
    return false;    // All other exceptions will get rethrown
  });
}
```

Colecciones Concurrentes

.NET ofrece colecciones seguras para hilos en el espacio de nombres System.Collections.Concurrent:

Concurrent collection	Nonconcurrent equivalent
ConcurrentStack<T>	Stack<T>
ConcurrentQueue<T>	Queue<T>
ConcurrentBag<T>	(none)
ConcurrentDictionary<TKey,TValue>	Dictionary<TKey,TValue>

Las colecciones concurrentes están optimizadas para escenarios de alta concurrencia; sin embargo, también pueden ser útiles cuando se necesita una colección segura para hilos (como alternativa al bloqueo alrededor de una colección ordinaria). Sin embargo, hay algunas advertencias:

- Las colecciones convencionales superan en rendimiento a las colecciones concurrentes, excepto en escenarios de alta concurrencia.
- Una colección segura para hilos no garantiza que el código que la utiliza sea seguro para hilos (ver "Bloqueo y seguridad para hilos" en la página 874).

- Si se enumera una colección concurrente mientras otro hilo la está modificando, no se produce una excepción; en su lugar, se obtiene una mezcla de contenido antiguo y nuevo.
- No hay una versión concurrente de List<T>.
- Las clases de pila, cola y bolsa concurrentes se implementan internamente con listas enlazadas. Esto las hace menos eficientes en términos de memoria que las clases Stack y Queue no concurrentes, pero mejores para el acceso concurrente porque las listas enlazadas son propicias para implementaciones sin bloqueo o con bajo bloqueo (esto se debe a que insertar un nodo en una lista enlazada requiere actualizar solo un par de referencias, mientras que insertar un elemento en una estructura similar a List<T> puede requerir mover miles de elementos existentes).

En otras palabras, estas colecciones no son simplemente atajos para usar una colección ordinaria con un bloqueo. Para demostrarlo, si ejecutamos el siguiente código en un solo hilo:

```
var d = new ConcurrentDictionary<int,int>();
for (int i = 0; i < 1000000; i++) d[i] = 123;
```

se ejecuta tres veces más lento que esto:

```
var d = new Dictionary<int,int>();
for (int i = 0; i < 1000000; i++) lock (d) d[i] = 123;
```

(Sin embargo, la lectura de un ConcurrentDictionary es rápida porque las lecturas no requieren bloqueo).

Las colecciones concurrentes también difieren de las colecciones convencionales en que exponen métodos especiales para realizar operaciones atómicas de prueba y actuación, como TryPop. La mayoría de estos métodos se unifican a través de la interfaz IProducerConsumerCollection<T>.

IProducerConsumerCollection<T>

Una colección productor/consumidor es aquella en la que los dos casos de uso principales son:

- Agregar un elemento ("producir").
- Recuperar un elemento mientras se elimina ("consumir").

Los ejemplos clásicos son las pilas (stacks) y las colas (queues). Las colecciones productor/consumidor son importantes en la programación paralela porque son propicias para implementaciones eficientes sin bloqueo.

La interfaz IProducerConsumerCollection<T> representa una colección productor/consumidor segura para hilos. Las siguientes clases implementan esta interfaz:

```
ConcurrentStack<T>
ConcurrentQueue<T>
ConcurrentBag<T>
```

IProducerConsumerCollection<T> extiende ICollection, y agrega los siguientes métodos:

```
void CopyTo (T[] array, int index);
T[] ToArray();
bool TryAdd (T item);
bool TryTake (out T item);
```

Los métodos TryAdd y TryTake prueban si se puede realizar una operación de agregado/eliminación; si es así, realizan la operación. La prueba y la acción se realizan atómicamente, lo que elimina la necesidad de bloquear como se haría alrededor de una colección convencional:

```
int result;
lock (myStack) if (myStack.Count > 0) result = myStack.Pop();
```

TryTake devuelve false si la colección está vacía. TryAdd siempre tiene éxito y devuelve true en las tres implementaciones proporcionadas. Sin embargo, si escribieras tu propia colección concurrente que prohibiera duplicados, harías que TryAdd devolviera false si el elemento ya existía (un ejemplo sería si escribieras un conjunto concurrente).

El elemento particular que TryTake elimina está definido por la subclase:

- Con una pila (stack), TryTake elimina el elemento más recientemente agregado.
- Con una cola (queue), TryTake elimina el elemento menos recientemente agregado.
- Con una bolsa (bag), TryTake elimina cualquier elemento que pueda eliminar de la manera más eficiente.

Las tres clases concretas implementan principalmente los métodos TryTake y TryAdd de forma explícita, exponiendo la misma funcionalidad a través de métodos públicos con nombres más específicos, como TryDequeue y TryPop.

ConcurrentBag<T>

ConcurrentBag<T> almacena una colección desordenada de objetos (se permiten duplicados). ConcurrentBag<T> es adecuada en situaciones en las que no importa qué elemento se obtenga al llamar a Take o TryTake.

El beneficio de ConcurrentBag<T> en comparación con una cola (queue) o una pila (stack) concurrente es que el método Add de una bolsa (bag) sufre casi ninguna contención cuando es llamado por muchos hilos al mismo tiempo. En contraste, llamar a Add en paralelo en una cola o una pila incurre en cierta contención (aunque mucho menos que bloquear alrededor de una colección no concurrente). Llamar a Take en una bolsa concurrente también es muy eficiente, siempre y cuando cada hilo no tome más elementos de los que agregó.

Dentro de una bolsa concurrente, cada hilo obtiene su propia lista enlazada privada. Los elementos se agregan a la lista privada que pertenece al hilo que llama a Add, eliminando la contención. Cuando se itera sobre la bolsa, el enumerador recorre la lista privada de cada hilo, devolviendo cada uno de sus elementos sucesivamente.

Cuando llamas a Take, la bolsa primero revisa la lista privada del hilo actual. Si hay al menos un elemento, puede completar la operación fácilmente y sin contención. Pero si la lista está vacía, debe "robar" un elemento de la lista privada de otro hilo e incurrir en la posibilidad de contención.

Entonces, para ser precisos, llamar a Take te devuelve el elemento agregado más recientemente en ese hilo; si no hay elementos en ese hilo, te devuelve el elemento agregado más recientemente en otro hilo, elegido al azar.

Las bolsas concurrentes son ideales cuando la operación paralela en tu colección consiste principalmente en agregar elementos, o cuando las operaciones de agregar y tomar están equilibradas en un hilo.

Vimos un ejemplo de lo primero anteriormente, al usar Parallel.ForEach para implementar un corrector ortográfico paralelo:

```
var misspellings = new ConcurrentBag<Tuple<int,string>>();

Parallel.ForEach (wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains (word))
    misspellings.Add (Tuple.Create ((int) i, word));
});
```

Una bolsa concurrente sería una elección inadecuada para una cola productor/consumidor, ya que los elementos se agregan y se eliminan por diferentes hilos.

BlockingCollection<T>

Si llamas a TryTake en cualquiera de las colecciones productor/consumidor que discutimos en la sección anterior, ConcurrentStack<T>, ConcurrentQueue<T> y ConcurrentBag<T>, y la colección está vacía, el método devuelve false. A veces, sería más útil en este escenario esperar hasta que haya un elemento disponible.

En lugar de sobrecargar los métodos TryTake con esta funcionalidad (lo cual habría causado una explosión de miembros después de permitir tokens de cancelación y tiempos de espera), los diseñadores de PFX encapsularon esta funcionalidad en una clase envoltorio llamada BlockingCollection<T>. Una blocking collection envuelve cualquier colección que implemente IProducerConsumerCollection<T> y te permite tomar un elemento de la colección envuelta, bloqueándote si no hay ningún elemento disponible.

Una blocking collection también te permite limitar el tamaño total de la colección, bloqueando al productor si se excede ese tamaño. Una colección limitada de esta manera se llama bounded blocking collection (colección de bloqueo limitada).

Para usar BlockingCollection<T>:

1. Instancia la clase, especificando opcionalmente la IProducerConsumerCollection<T> para envolver y el tamaño máximo (límite) de la colección.

2. Llama a Add o TryAdd para agregar elementos a la colección subyacente.

3. Llama a Take o TryTake para eliminar (consumir) elementos de la colección subyacente.

Si llamas al constructor sin pasar una colección, la clase automáticamente instanciará un ConcurrentQueue<T>. Los métodos de producción y consumo te permiten especificar tokens de cancelación y tiempos de espera. Add y TryAdd pueden bloquearse si el tamaño de la colección está limitado; Take y TryTake se bloquean mientras la colección está vacía.

Otra forma de consumir elementos es llamando a GetConsumingEnumerable. Esto devuelve una secuencia (potencialmente infinita) que proporciona elementos a medida que están disponibles. Puedes finalizar la secuencia llamando a CompleteAdding: este método también evita que se enlisten más elementos.

BlockingCollection también proporciona métodos estáticos llamados AddToAny y TakeFromAny, que te permiten agregar o tomar un elemento mientras especificas varias blocking collections. La acción se realiza en la primera colección capaz de atender la solicitud.

Escribir una cola productor/consumidor (Writing a Producer/Consumer Queue)

Una cola productor/consumidor es una estructura útil tanto en programación paralela como en escenarios generales de concurrencia. Así es como funciona:

- Se configura una cola para describir elementos de trabajo o datos sobre los que se realiza un trabajo.
- Cuando una tarea necesita ser ejecutada, se encola y el llamador continúa con otras tareas.
- Uno o más hilos de trabajo trabajan en segundo plano, tomando y ejecutando los elementos encolados.

Una cola productor/consumidor te brinda un control preciso sobre cuántos hilos de trabajo se ejecutan a la vez, lo cual es útil para limitar no solo el consumo de CPU, sino también otros recursos. Si las tareas realizan operaciones intensivas de E/S en disco, por ejemplo, puedes limitar la concurrencia para evitar que el sistema operativo y otras aplicaciones se queden sin recursos. También puedes agregar y eliminar dinámicamente trabajadores a lo largo de la vida de la cola.

El grupo de hilos (thread pool) del CLR en sí mismo es una especie de cola productor/consumidor, optimizada para trabajos de corta duración y con uso intensivo de la CPU.

Una cola productor/consumidor generalmente contiene elementos de datos sobre los cuales se realiza la misma tarea. Por ejemplo, los elementos de datos pueden ser nombres de archivos y la tarea podría ser cifrar esos archivos. Sin embargo, al hacer que el elemento sea un delegado, puedes escribir una cola productor/consumidor más general donde cada elemento puede hacer cualquier cosa.

En http://albahari.com/threading, mostramos cómo escribir una cola productor/consumidor desde cero utilizando un AutoResetEvent (y más tarde, utilizando Wait y Pulse de Monitor). Sin embargo, escribir una cola productor/consumidor desde cero no es necesario porque la mayor parte de la funcionalidad está provista por BlockingCollection<T>. Así es cómo se utiliza:

```
public class PCQueue : IDisposable
{
  BlockingCollection<Action> _taskQ = new BlockingCollection<Action>();

  public PCQueue (int workerCount)
  {
    // Create and start a separate Task for each consumer:
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew (Consume);
  }

  public void Enqueue (Action action) { _taskQ.Add (action); }

  void Consume()
  {
    // This sequence that we're enumerating will block when no elements
    // are available and will end when CompleteAdding is called.

    foreach (Action action in _taskQ.GetConsumingEnumerable())
      action();     // Perform task.
  }

  public void Dispose() { _taskQ.CompleteAdding(); }
}
```

Como no pasamos nada al constructor de BlockingCollection, se instanció automáticamente una cola concurrente. Si hubiéramos pasado un ConcurrentStack, habríamos obtenido una pila productor/consumidor.

Utilizando tareas (Using Tasks)

El productor/consumidor que acabamos de escribir es inflexible en el sentido de que no podemos rastrear los elementos de trabajo una vez que se han encolado. Sería bueno si pudiéramos hacer lo siguiente:

- Saber cuándo se ha completado un elemento de trabajo (y esperarlo)
- Cancelar un elemento de trabajo
- Manejar elegantemente cualquier excepción generada por un elemento de trabajo

Una solución ideal sería que el método Enqueue devolviera algún objeto que nos brinde la funcionalidad descrita anteriormente. La buena noticia es que ya existe una clase que hace exactamente esto: la clase Task, que podemos generar tanto con un TaskCompletionSource como instanciándola directamente (creando una tarea no iniciada o "fría"):

```
public class PCQueue : IDisposable
{
  BlockingCollection<Task> _taskQ = new BlockingCollection<Task>();

  public PCQueue (int workerCount)
  {
    // Create and start a separate Task for each consumer:
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew (Consume);
  }

  public Task Enqueue (Action action, CancellationToken cancelToken
                                          = default (CancellationToken))
  {
    var task = new Task (action, cancelToken);
    _taskQ.Add (task);
    return task;
  }

  public Task<TResult> Enqueue<TResult> (Func<TResult> func,
              CancellationToken cancelToken = default (CancellationToken))
  {
    var task = new Task<TResult> (func, cancelToken);
    _taskQ.Add (task);
    return task;
  }

  void Consume()
  {
    foreach (var task in _taskQ.GetConsumingEnumerable())
      try
      {
          if (!task.IsCanceled) task.RunSynchronously();
      }
      catch (InvalidOperationException) { }  // Race condition
  }

  public void Dispose() { _taskQ.CompleteAdding(); }
}
```

En el método Enqueue, encolamos y devolvemos al llamador una tarea que creamos pero no iniciamos.

En Consume, ejecutamos la tarea de forma sincrónica en el hilo del consumidor. Capturamos una InvalidOperationException para manejar el evento poco probable de que la tarea sea cancelada entre verificar si está cancelada y ejecutarla.

Así es cómo podemos utilizar esta clase:

```
var pcQ = new PCQueue (2);     // Maximum concurrency of 2
string result = await pcQ.Enqueue (() => "That was easy!");
...
```

De esta manera, obtenemos todos los beneficios de las tareas, como la propagación de excepciones, los valores de retorno y la cancelación, al mismo tiempo que tenemos un control completo sobre la planificación.

23 Span<T> y Memory<T>

Span<T> y Memory<T> son estructuras que actúan como fachadas de bajo nivel sobre una matriz (array), cadena (string) o cualquier bloque contiguo de memoria administrada o no administrada. Su principal objetivo es ayudar con ciertos tipos de microoptimizaciones, en particular, escribir código de baja asignación de memoria que minimice las asignaciones de memoria administrada (y así reducir la carga en el recolector de basura) sin tener que duplicar el código para diferentes tipos de entrada. También permiten realizar divisiones (slicing), es decir, trabajar con una porción de una matriz, cadena o bloque de memoria sin crear una copia.

Span<T> y Memory<T> son particularmente útiles en áreas críticas de rendimiento, como el canal de procesamiento de ASP.NET Core o un analizador JSON que sirve como base de datos de objetos.



Si te encuentras con estos tipos en una API y no necesitas ni te importan sus posibles ventajas de rendimiento, puedes tratarlos fácilmente de la siguiente manera:

- Al llamar a un método que espera un Span<T>, ReadOnly Span<T>, Memory<T> o ReadOnlyMemory<T>, pasa una matriz (array) en su lugar, es decir, T[]. (Esto funciona gracias a los operadores de conversión implícita).
- Para convertir un span/memory a una matriz, llama al método ToArray. Y si T es char, ToString convertirá el span/memory en una cadena (string).

Específicamente, Span<T> hace dos cosas:

- Proporciona una interfaz común similar a una matriz (array) sobre matrices administradas, cadenas y memoria respaldada por punteros. Esto te brinda la libertad de utilizar memoria asignada en la pila (stack-allocated) y memoria no administrada para evitar la recolección de basura, sin tener que duplicar el código o manipular punteros.
- Permite "slicing": exponer subsecciones reutilizables del span sin hacer copias.



Span<T> consta de solo dos campos, un puntero y una longitud. Por esta razón, solo puede representar bloques de memoria contiguos. (Si necesitas trabajar con memoria no contigua, la clase ReadOnlySequence<T> está disponible para servir como una lista enlazada).

Debido a que Span<T> puede envolver memoria asignada en la pila (stack-allocated), existen restricciones en cómo puedes almacenar o pasar instancias (impuestas, en parte, por Span<T> siendo una ref struct). Memory<T> actúa como un span sin esas restricciones, pero no puede envolver memoria asignada en la pila. Memory<T> aún proporciona la ventaja de realizar divisiones (slicing).

Cada estructura viene con su contraparte de solo lectura (ReadOnlySpan<T> y ReadOnlyMemory<T>). Además de evitar cambios no deseados, las contrapartes de solo lectura mejoran aún más el rendimiento al permitirle al compilador y al tiempo de ejecución una mayor libertad para la optimización.

.NET en sí mismo (y ASP.NET Core) utilizan estos tipos para mejorar la eficiencia con E/S, manejo de redes, manipulación de cadenas y análisis de JSON.


La capacidad de Span<T> y Memory<T> para realizar divisiones de matrices hace que la antigua clase ArraySegment<T> sea redundante. Para ayudar en cualquier transición, existen operadores de conversión implícitos desde ArraySegment<T> a todas las estructuras de span/memory, y desde Memory<T> y ReadOnlyMemory<T> a ArraySegment<T>.

Spans y Slicing

Supongamos que estás escribiendo un método para sumar una matriz de enteros. Una implementación microoptimizada evitaría LINQ a favor de un bucle foreach:

```
int Sum (int[] numbers)
{
  int total = 0;
  foreach (int i in numbers) total += i;
  return total;
}
```

Ahora imagina que quieres sumar solo una porción de la matriz. Tienes dos opciones:

- Copiar la porción de la matriz que deseas sumar a otra matriz.
- Agregar parámetros adicionales (offset y count).

La primera opción es ineficiente; la segunda opción agrega desorden y complejidad (lo cual empeora con métodos que necesitan aceptar más de una matriz).

Los spans resuelven esto de manera elegante. Lo único que tienes que hacer es cambiar el tipo del parámetro de int[] a ReadOnlySpan<int> (todo lo demás permanece igual):

```
int Sum (ReadOnlySpan<int> numbers)
{
  int total = 0;
  foreach (int i in numbers) total += i;
  return total;
}
```


Utilizamos ReadOnlySpan<T> en lugar de Span<T> porque no necesitamos modificar la matriz. Existe una conversión implícita de Span<T> a ReadOnlySpan<T>, por lo que puedes pasar un Span<T> a un método que espera un ReadOnlySpan<T>.

Podemos probar este método de la siguiente manera:

```
var numbers = new int [1000];
for (int i = 0; i < numbers.Length; i++) numbers [i] = i;

int total = Sum (numbers);
```

Podemos llamar a Sum con una matriz porque hay una conversión implícita de T[] a Span<T> y ReadOnlySpan<T>. Otra opción es usar el método de extensión AsSpan():

```
var span = numbers.AsSpan();
```

El indexador de ReadOnlySpan<T> utiliza la característica "ref readonly" de C# para acceder directamente a los datos subyacentes: esto permite que nuestro método funcione casi tan bien como el ejemplo original que utilizaba una matriz. Pero lo que hemos ganado es que ahora podemos "cortar" la matriz y sumar solo una parte de los elementos de la siguiente manera:

```
// Sum the middle 500 elements (starting from position 250):
int total = Sum (numbers.AsSpan (250, 500));
```

Si ya tienes un Span<T> o ReadOnlySpan<T>, puedes cortarlo llamando al método Slice:

```
Span<int> span = numbers;
int total = Sum (span.Slice (250, 500));
```

También puedes usar los índices y rangos de C# (a partir de C# 8):

```
Span<int> span = numbers;
Console.WriteLine (span [^1]);            // Last element
Console.WriteLine (Sum (span [..10]));    // First 10 elements
Console.WriteLine (Sum (span [100..]));   // 100th element to end
Console.WriteLine (Sum (span [^5..]));    // Last 5 elements
```

Aunque Span<T> no implementa IEnumerable<T> (no puede implementar interfaces debido a ser una ref struct), implementa el patrón que permite que la instrucción foreach de C# funcione (ver "Enumeración" en la página 189).

CopyTo y TryCopyTo

El método CopyTo copia elementos de un span (o Memory<T>) a otro. En el siguiente ejemplo, copiamos todos los elementos del span x en el span y:

```
Span<int> x = new[] { 1, 2, 3, 4 };
Span<int> y = new int[4];
x.CopyTo (y);
```

El uso de la función de corte (slicing) hace que este método sea mucho más útil. En el siguiente ejemplo, copiamos la primera mitad del span x en la segunda mitad del span y:

```
Span<int> x = new[] { 1,  2,  3,  4  };
Span<int> y = new[] { 10, 20, 30, 40 };
x[..2].CopyTo (y[2..]);                // y is now { 10, 20, 1, 2 }
```

Si no hay suficiente espacio en el destino para completar la copia, CopyTo lanzará una excepción, mientras que TryCopyTo devolverá false (sin copiar ningún elemento).

Las estructuras de span también tienen métodos para borrar (Clear) y llenar (Fill) el span, así como un método IndexOf para buscar un elemento en el span.

Trabajar con texto

Los spans están diseñados para funcionar bien con las cadenas de texto, que se tratan como ReadOnlySpan<char>. El siguiente método cuenta los caracteres de espacio en blanco:

```
int CountWhitespace (ReadOnlySpan<char> s)
{
  int count = 0;
  foreach (char c in s)
    if (char.IsWhiteSpace (c))
      count++;
  return count;
}
```

Puedes llamar a este método con una cadena de texto (gracias a un operador de conversión implícito):

```
int x = CountWhitespace ("Word1 Word2");   // OK
```

o con una subcadena:

```
int y = CountWhitespace (someString.AsSpan (20, 10));
```

El método ToString() convierte un ReadOnlySpan<char> de vuelta a una cadena de texto.

Los métodos de extensión aseguran que algunos de los métodos comúnmente utilizados de la clase string también estén disponibles para ReadOnlySpan<char>:

```
var span = "This ".AsSpan();                    // ReadOnlySpan<char>
Console.WriteLine (span.StartsWith ("This"));   // True
Console.WriteLine (span.Trim().Length);         // 4
```

(Ten en cuenta que los métodos como StartsWith utilizan una comparación ordinal, mientras que los métodos correspondientes de la clase string utilizan una comparación sensible a la cultura de forma predeterminada).

Los métodos como ToUpper y ToLower están disponibles, pero debes pasar un span de destino con la longitud correcta (esto te permite decidir cómo y dónde asignar la memoria).

Algunos de los métodos de la clase string no están disponibles, como Split (que divide una cadena en un array de palabras). En realidad, es imposible escribir el equivalente directo del método Split de string porque no se puede crear un array de spans.



Esto se debe a que los spans se definen como ref structs, que solo pueden existir en la pila (stack).

(Cuando decimos "existen solo en la pila", nos referimos a que la estructura en sí solo puede existir en la pila. El contenido que el span envuelve puede, y de hecho lo hace en este caso, existir en el montón (heap).)

El espacio de nombres System.Buffers.Text contiene tipos adicionales que te ayudan a trabajar con texto basado en spans, incluyendo lo siguiente:

- Utf8Formatter.TryFormat realiza el equivalente de llamar a ToString en tipos integrados y simples como decimal, DateTime, etc., pero escribe en un span en lugar de una cadena.
- Utf8Parser.TryParse hace lo contrario y analiza datos desde un span hacia un tipo simple.
- El tipo Base64 proporciona métodos para leer/escribir datos en base64.

Métodos fundamentales del CLR, como int.Parse, también han sido sobrecargados para aceptar ReadOnlySpan<char>.

Memory<T>

Span<T> y ReadOnlySpan<T> se definen como ref structs para maximizar su potencial de optimización y permitirles trabajar de forma segura con memoria asignada en la pila (como verás en la siguiente sección). Sin embargo, también imponen limitaciones. Además de no ser compatibles con los arrays, no se pueden utilizar como campos en una clase (esto los colocaría en el montón). Esto, a su vez, impide que aparezcan en expresiones lambda, así como parámetros en métodos asíncronos, iteradores y flujos asíncronos:

```
async void Foo (Span<int> notAllowed)   // Compile-time error!
```

(Recuerda que el compilador procesa los métodos asíncronos y los iteradores escribiendo una máquina de estados privada, lo que significa que cualquier parámetro y variable local terminan como campos. Lo mismo se aplica a las expresiones lambda que cierran sobre variables: también terminan como campos en un closure).

Las estructuras Memory<T> y ReadOnlyMemory<T> solucionan esto, actuando como spans que no pueden envolver memoria asignada en la pila, lo que permite su uso en campos, expresiones lambda, métodos asíncronos, etc.

Puedes obtener un Memory<T> o ReadOnlyMemory<T> desde un array mediante una conversión implícita o mediante el método de extensión AsMemory():

```
Memory<int> mem1 = new int[] { 1, 2, 3 };
var mem2 = new int[] { 1, 2, 3 }.AsMemory();
```

Es fácil "convertir" un Memory<T> o ReadOnlyMemory<T> en un Span<T> o ReadOnlySpan<T> a través de su propiedad Span, de modo que puedas interactuar con él como si fuera un span. La conversión es eficiente en el sentido de que no realiza ninguna copia:

```
async void Foo (Memory<int> memory)
{
  Span<int> span = memory.Span;
  ...
}
```

(También puedes cortar directamente un Memory<T> o ReadOnlyMemory<T> mediante su método Slice o un rango de C#, y acceder a su longitud mediante su propiedad Length).



Otra forma de obtener un Memory<T> es tomarlo prestado de un pool utilizando la clase System.Buffers.MemoryPool<T>. Esto funciona de manera similar al agrupamiento de arrays y ofrece otra estrategia para reducir la carga en el recolector de basura.

Dijimos en la sección anterior que no se puede escribir el equivalente directo de string.Split para spans, porque no se puede crear un array de spans. Esta limitación no se aplica a ReadOnlyMemory<char>:

```
// Split a string into words:
IEnumerable<ReadOnlyMemory<char>> Split (ReadOnlyMemory<char> input)
{
  int wordStart = 0;
  for (int i = 0; i <= input.Length; i++)
    if (i == input.Length || char.IsWhiteSpace (input.Span [i]))
    {
      yield return input [wordStart..i];   // Slice with C# range operator
      wordStart = i + 1;
    }
}
```

Esto es más eficiente que el método Split de string: en lugar de crear nuevas cadenas para cada palabra, devuelve segmentos de la cadena original.

```
foreach (var slice in Split ("The quick brown fox jumps over the lazy dog"))
{
  // slice is a ReadOnlyMemory<char>
}
```



Es fácil convertir un Memory<T> en un Span<T> mediante la propiedad Span, pero no al revés. Por esta razón, es mejor escribir métodos que acepten Span<T> en lugar de Memory<T> cuando tengas opción.

Por la misma razón, es mejor escribir métodos que acepten ReadOnlySpan<T> en lugar de Span<T>.

Enumeradores de solo avance

En la sección anterior, utilizamos ReadOnlyMemory<char> como una solución para implementar un método Split similar al de las cadenas de texto. Sin embargo, al renunciar a ReadOnlySpan<char>, perdimos la capacidad de dividir spans respaldados por memoria no

administrada. Vamos a volver a examinar ReadOnlySpan<char> para ver si podemos encontrar otra solución.

Una posible opción sería escribir nuestro método Split para que devuelva rangos (Range):

```
Range[] Split (ReadOnlySpan<char> input)
{
  int pos = 0;
  var list = new List<Range>();
  for (int i = 0; i <= input.Length; i++)
    if (i == input.Length || char.IsWhiteSpace (input [i]))
    {
      list.Add (new Range (pos, i));
      pos = i + 1;
    }
  return list.ToArray();
}
```

El llamador podría utilizar esos rangos para dividir el span original:

```
ReadOnlySpan<char> source = "The quick brown fox";
foreach (Range range in Split (source))
{
  ReadOnlySpan<char> wordSpan = source [range];
  ...
}
```

Esto es una mejora, pero aún no es perfecto. Una de las razones para usar spans es evitar las asignaciones de memoria. Sin embargo, observa que nuestro método Split crea una lista (List<Range>), agrega elementos a ella y luego convierte la lista en un array. Esto implica al menos dos asignaciones de memoria y una operación de copia de memoria.

La solución a esto es evitar la lista y el array a favor de un enumerador de solo avance (forward-only enumerator). Un enumerador es más complicado de usar, pero se puede hacer sin asignaciones de memoria utilizando structs:

```
// We must define this as a ref struct, because _input is a ref struct.
public readonly ref struct CharSpanSplitter
{
  readonly ReadOnlySpan<char> _input;
  public CharSpanSplitter (ReadOnlySpan<char> input) => _input = input;
  public Enumerator GetEnumerator() => new Enumerator (_input);

  public ref struct Enumerator   // Forward-only enumerator
  {
    readonly ReadOnlySpan<char> _input;
    int _wordPos;
    public ReadOnlySpan<char> Current { get; private set; }

    public Rator (ReadOnlySpan<char> input)
```

```
    {
      _input = input;
      _wordPos = 0;
      Current = default;
    }

    public bool MoveNext()
    {
      for (int i = _wordPos; i <= _input.Length; i++)
        if (i == _input.Length || char.IsWhiteSpace (_input [i]))
        {
          Current = _input [_wordPos..i];
          _wordPos = i + 1;
          return true;
        }
      return false;
    }
  }
}

public static class CharSpanExtensions
{
  public static CharSpanSplitter Split (this ReadOnlySpan<char> input)
    => new CharSpanSplitter (input);

  public static CharSpanSplitter Split (this Span<char> input)
    => new CharSpanSplitter (input);
}
```

Aquí tienes cómo se llamaría:

```
var span = "the quick brown fox".AsSpan();
foreach (var word in span.Split())
{
  // word is a ReadOnlySpan<char>
}
```

Al definir una propiedad Current y un método MoveNext, nuestro enumerador puede funcionar con la instrucción foreach de C# (ver "Enumeración" en la página 189). No tenemos que implementar las interfaces IEnumerable<T>/IEnumerator<T> (de hecho, no podemos hacerlo; las ref structs no pueden implementar interfaces). Estamos sacrificando la abstracción por una micro-optimización.

Trabajar con memoria asignada en la pila y memoria no administrada

Otra técnica de micro-optimización efectiva es reducir la carga en el recolector de basura al minimizar las asignaciones en el montón (heap). Esto significa hacer un mayor uso de la memoria basada en la pila o incluso en memoria no administrada.

Desafortunadamente, esto normalmente requiere reescribir el código para usar punteros. En el caso de nuestro ejemplo anterior que sumaba elementos en un array, tendríamos que escribir otra versión de la siguiente manera:

```
unsafe int Sum (int* numbers, int length)
{
  int total = 0;
  for (int i = 0; i < length; i++) total += numbers [i];
  return total;
}
```

para poder hacer esto:

```
int* numbers = stackalloc int [1000]; // Allocate array on the stack
int total = Sum (numbers, 1000);
```

Las Spans resuelven este problema: puedes construir una Span<T> o ReadOnlySpan<T> directamente desde un puntero:

```
int* numbers = stackalloc int [1000];
var span = new Span<int> (numbers, 1000);
```

O en un solo paso:

```
Span<int> numbers = stackalloc int [1000];
```

(Ten en cuenta que esto no requiere el uso de "unsafe".) Recuerda el método Sum que escribimos anteriormente:

```
int Sum (ReadOnlySpan<int> numbers)
{
  int total = 0;
  int len = numbers.Length;
  for (int i = 0; i < len; i++) total += numbers [i];
  return total;
}
```

Este método funciona igual de bien para un span asignado en la pila. Hemos ganado en tres aspectos:

- El mismo método funciona tanto con arrays como con memoria asignada en la pila.
- Podemos usar memoria asignada en la pila con un uso mínimo de punteros.
- El span puede ser dividido en sub-spans.



El compilador es lo suficientemente inteligente como para evitar que escribas un método que asigna memoria en la pila y la devuelve al llamador a través de un Span<T> o ReadOnlySpan<T>.

(Sin embargo, en otros escenarios, puedes devolver legalmente un Span<T> o ReadOnlySpan<T>.)

También puedes usar spans para envolver memoria que asignas desde el montón no administrado. En el siguiente ejemplo, asignamos memoria no administrada usando la función Marshal.AllocHGlobal, la envolvemos en un Span<char> y luego copiamos una cadena en la memoria no administrada. Finalmente, utilizamos la estructura CharSpanSplitter que escribimos en la sección anterior para dividir la cadena no administrada en palabras:

```
var source = "The quick brown fox".AsSpan();
var ptr = Marshal.AllocHGlobal (source.Length * sizeof (char));
try
{
  var unmanaged = new Span<char> ((char*)ptr, source.Length);
  source.CopyTo (unmanaged);
  foreach (var word in unmanaged.Split())
    Console.WriteLine (word.ToString());
}
finally { Marshal.FreeHGlobal (ptr); }
```

Un bono adicional es que el indexador de Span<T> realiza verificación de límites (bounds-checking), evitando un desbordamiento del búfer. Esta protección se aplica si instancias correctamente Span<T>. En nuestro ejemplo, perderías esta protección si obtuvieras incorrectamente el span de la siguiente manera:

```
var span = new Span<char> ((char*)ptr, source.Length * 2);
```

Tampoco hay protección contra el equivalente de un puntero colgante, por lo que debes tener cuidado de no acceder al span después de liberar su memoria no administrada con Marshal.FreeHGlobal.

24 Interoperabilidad nativa y COM

Este capítulo describe cómo integrarse con bibliotecas de vínculos dinámicos (DLL) nativas (no administradas) y componentes del Component Object Model (COM). A menos que se indique lo contrario, los tipos mencionados en este capítulo existen en los espacios de nombres System o System.Runtime.InteropServices.

Llamando a DLLs nativas

P/Invoke, que significa Platform Invocation Services, te permite acceder a funciones, estructuras y devoluciones de llamada en DLLs no administradas (bibliotecas compartidas en Unix).

Por ejemplo, considera la función MessageBox, definida en la DLL de Windows user32.dll, de la siguiente manera:

```
int MessageBox (HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

Puedes llamar a esta función directamente declarando un método estático con el mismo nombre, aplicando la palabra clave extern y agregando el atributo DllImport:

```
using System;
using System.Runtime.InteropServices;

MessageBox (IntPtr.Zero,
            "Please do not press this again.", "Attention", 0);

[DllImport("user32.dll")]
static extern int MessageBox (IntPtr hWnd, string text, string caption,
                              int type);
```

Las clases MessageBox en los espacios de nombres System.Windows y System.Windows.Forms en sí llaman a métodos no administrados similares.

Aquí tienes un ejemplo de DllImport para Ubuntu Linux:

```
Console.WriteLine ($"User ID: {getuid()}");

[DllImport("libc")]
static extern uint getuid();
```

El CLR incluye un marshaler que sabe cómo convertir parámetros y valores de retorno entre tipos de .NET y tipos no administrados. En el ejemplo de Windows, los parámetros int se traducen directamente a enteros de cuatro bytes que la función espera, y los parámetros de cadena se convierten en matrices de caracteres Unicode terminadas en nulo (codificadas en UTF-16). IntPtr es una estructura diseñada para encapsular un identificador no administrado; tiene 32 bits de ancho en plataformas de 32 bits y 64 bits de ancho en plataformas de 64 bits. Ocurre una traducción similar en Unix. (A partir de C# 9, también puedes usar el tipo nint, que se asigna a IntPtr).

Marshaling de tipos y parámetros

Marshalizando tipos comunes

En el lado no administrado, puede haber más de una forma de representar un tipo de datos dado. Por ejemplo, una cadena puede contener caracteres ANSI de un solo byte o caracteres Unicode UTF-16, y puede tener una longitud prefijada, estar terminada en nulo o tener una longitud fija. Con el atributo MarshalAs, puedes especificar al marshaler del CLR la variación que se está utilizando para que pueda proporcionar la traducción correcta. Aquí tienes un ejemplo:

```
[DllImport("...")]
static extern int Foo ( [MarshalAs (UnmanagedType.LPStr)] string s );
```

La enumeración UnmanagedType incluye todos los tipos Win32 y COM que el marshaler comprende. En este caso, se le indicó al marshaler que tradujera a LPStr, que es una cadena ANSI de un solo byte terminada en nulo.

En el lado de .NET, también tienes algunas opciones sobre qué tipo de datos usar. Por ejemplo, los identificadores no administrados pueden asignarse a IntPtr, int, uint, long o ulong.



La mayoría de los identificadores no administrados encapsulan una dirección o puntero y, por lo tanto, deben asignarse a IntPtr para ser compatibles con sistemas operativos de 32 y 64 bits. Un ejemplo típico es HWND.

Muy a menudo, con las funciones Win32 y POSIX, te encuentras con un parámetro entero que acepta un conjunto de constantes definidas en un archivo de encabezado de C++ como WinUser.h. En lugar de definir estas constantes como simples constantes de C#, puedes definirlas dentro de una enumeración. El uso de una enumeración puede proporcionar un código más ordenado y aumentar la seguridad de tipos estática. Proporcionamos un ejemplo en "Memoria compartida" en la página 971.



Cuando instales Microsoft Visual Studio, asegúrate de instalar los archivos de encabezado de C++ aunque elijas nada más en la categoría de C++. Ahí es donde se definen todas las constantes nativas de Win32. Luego puedes localizar todos los archivos de encabezado buscando "*.h" en el directorio del programa de Visual Studio.

En Unix, el estándar POSIX define los nombres de las constantes, pero las implementaciones individuales de sistemas Unix compatibles con POSIX pueden asignar diferentes valores numéricos a estas constantes. Debes usar el valor numérico correcto para el sistema operativo que elijas. De manera similar, POSIX define un estándar para las estructuras utilizadas en llamadas de interoperabilidad. El orden de los campos en la estructura no está fijado por el estándar, y una implementación Unix podría agregar campos adicionales. Los archivos de encabezado de C++ que definen funciones y tipos a menudo se instalan en /usr/include o /usr/local/include.

Recibir cadenas de código no administrado de vuelta a .NET requiere que se realice una gestión de memoria. El marshaler realiza automáticamente este trabajo si declaras el método externo con un StringBuilder en lugar de una cadena, de la siguiente manera:

```
StringBuilder s = new StringBuilder (256);
GetWindowsDirectory (s, 256);
Console.WriteLine (s);

[DllImport("kernel32.dll")]
static extern int GetWindowsDirectory (StringBuilder sb, int maxChars);
```

En Unix, funciona de manera similar. Las siguientes llamadas a getcwd devuelven el directorio actual:

```
var sb = new StringBuilder (256);
Console.WriteLine (getcwd (sb, sb.Capacity));

[DllImport("libc")]
static extern string getcwd (StringBuilder buf, int size);
```

Aunque StringBuilder es conveniente de usar, es algo ineficiente porque el CLR debe realizar asignaciones y copias de memoria adicionales. En puntos críticos de rendimiento, puedes evitar esta sobrecarga utilizando char[] en su lugar:

```
[DllImport ("kernel32.dll", CharSet = CharSet.Unicode)]
static extern int GetWindowsDirectory (char[] buffer, int maxChars);
```

Observa que debes especificar un CharSet en el atributo DllImport. También debes recortar la cadena de salida a su longitud después de llamar a la función. Puedes lograr esto minimizando las asignaciones de memoria mediante el uso de agrupación de matrices (ver "Agrupación de matrices" en la página 575), de la siguiente manera:

```
string GetWindowsDirectory()
{
  var array = ArrayPool<char>.Shared.Rent (256);
  try
  {
    int length = GetWindowsDirectory (array, 256);

     return new string (array, 0, length).ToString();
   }
   finally { ArrayPool<char>.Shared.Return (array); }
 }
```

(Por supuesto, este ejemplo es ficticio, ya que puedes obtener el directorio de Windows a través del método integrado Environment.GetFolderPath.)



Si no estás seguro de cómo llamar a un método específico de Win32 o Unix, generalmente encontrarás un ejemplo en Internet si buscas el nombre del método y DllImport. Para Windows, el sitio http://www.pinvoke.net es una wiki que tiene como objetivo documentar todas las firmas de Win32.

Interoperabilidad con Clases y Estructuras

A veces, necesitas pasar una estructura a un método no administrado. Por ejemplo, GetSystemTime en la API de Win32 se define de la siguiente manera:

```
void GetSystemTime (LPSYSTEMTIME lpSystemTime);
```

LPSYSTEMTIME se ajusta a esta estructura C:

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Para llamar a GetSystemTime, debemos definir una clase o estructura de .NET que coincida con esta estructura C:

```
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
class SystemTime
{
   public ushort Year;
   public ushort Month;
   public ushort DayOfWeek;
   public ushort Day;
   public ushort Hour;
   public ushort Minute;
   public ushort Second;
   public ushort Milliseconds;
}
```

El atributo StructLayout instruye al marshaler sobre cómo mapear cada campo a su equivalente sin administración de memoria. LayoutKind.Sequential significa que queremos que los campos se alineen secuencialmente en los límites del tamaño de empaquetado (verás lo que esto significa en breve), tal como lo harían en una estructura de C. Los nombres de los campos aquí no son relevantes; lo importante es el orden de los campos.

Ahora podemos llamar a GetSystemTime:

```
SystemTime t = new SystemTime();
GetSystemTime (t);
Console.WriteLine (t.Year);

[DllImport("kernel32.dll")]
static extern void GetSystemTime (SystemTime t);
```

De manera similar, en Unix:

```
Console.WriteLine (GetSystemTime());

static DateTime GetSystemTime()
{
  DateTime startOfUnixTime =
    new DateTime(1970, 1, 1, 0, 0, 0, 0, System.DateTimeKind.Utc);

  Timespec tp = new Timespec();
  int success = clock_gettime (0, ref tp);
  if (success != 0) throw new Exception ("Error checking the time.");
  return startOfUnixTime.AddSeconds (tp.tv_sec).ToLocalTime();
}

[DllImport("libc")]
static extern int clock_gettime (int clk_id, ref Timespec tp);

[StructLayout(LayoutKind.Sequential)]
struct Timespec
{
  public long tv_sec;   /* seconds */
  public long tv_nsec;  /* nanoseconds */
}
```

Tanto en C como en C#, los campos de un objeto se encuentran a n bytes de la dirección de ese objeto. La diferencia radica en que en un programa de C#, el CLR encuentra este desplazamiento buscándolo utilizando el token del campo; los nombres de los campos de C se compilan directamente en desplazamientos. Por ejemplo, en C, wDay es solo un token que representa lo que sea que esté en la dirección de una instancia de SystemTime más 24 bytes.

Para obtener velocidad de acceso, cada campo se coloca en un desplazamiento que es un múltiplo del tamaño del campo. Sin embargo, ese multiplicador está limitado a un máximo de x bytes, donde x es el tamaño de empaquetado. En la implementación actual, el tamaño de empaquetado predeterminado es de 8 bytes, por lo que una estructura que comprende un sbyte seguido de un long (de 8 bytes) ocupa 16 bytes, y los 7 bytes siguientes al sbyte se desperdician. Puedes reducir o eliminar este desperdicio especificando un tamaño de empaquetado a través de la propiedad Pack del atributo StructLayout: esto hace que los campos se alineen en desplazamientos que son múltiplos del tamaño de empaquetado especificado. Entonces, con un tamaño de empaquetado (pack size) de 1, la estructura descrita ocuparía solo 9 bytes. Puedes especificar tamaños de empaquetado de 1, 2, 4, 8 o 16 bytes.

El atributo StructLayout también te permite especificar desplazamientos explícitos de campos (ver "Simulating a C Union" en la página 970).

Marshaling de entrada y salida

En el ejemplo anterior, implementamos SystemTime como una clase. También podríamos haber elegido una estructura siempre que GetSystemTime se declarara con un parámetro ref o out:

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime (out SystemTime t);
```

En la mayoría de los casos, la semántica direccional de los parámetros de C# funciona de la misma manera con métodos externos. Los parámetros pasados por valor se copian al entrar, los parámetros ref de C# se copian al entrar/salir, y los parámetros out de C# se copian al salir. Sin embargo, hay algunas excepciones para los tipos que tienen conversiones especiales. Por ejemplo, las clases de matrices (arrays) y la clase StringBuilder requieren copia al salir de una función, por lo que se consideran parámetros de entrada/salida (in/out). Ocasionalmente, es útil anular este comportamiento con los atributos In y Out. Por ejemplo, si se desea que una matriz sea de solo lectura, el modificador in indica que solo se copie la matriz al entrar en la función, no al salir de ella:

```
static extern void Foo ( [In] int[] array);
```

Convenciones de llamada

Los métodos sin administración de memoria reciben argumentos y valores de retorno a través de la pila y (opcionalmente) registros de la CPU. Debido a que hay más de una forma de lograr esto, han surgido varios protocolos diferentes. Estos protocolos se conocen como convenciones de llamada (calling conventions).

El CLR actualmente admite tres convenciones de llamada: StdCall, Cdecl y ThisCall. De forma predeterminada, el CLR utiliza la convención de llamada predeterminada de la plataforma (la convención estándar para esa plataforma). En Windows, es StdCall, y en Linux x86, es Cdecl.

Si un método sin administración de memoria no sigue esta convención predeterminada, puedes indicar explícitamente su convención de llamada de la siguiente manera:

```
[DllImport ("MyLib.dll", CallingConvention=CallingConvention.Cdecl)]
static extern void SomeFunc (...)
```

El nombre CallingConvention.WinApi, que puede resultar un poco confuso, se refiere a la convención predeterminada de la plataforma.

Devolver llamadas desde código no administrado

C# también permite que las funciones externas llamen al código C# mediante devoluciones de llamada (callbacks). Hay dos formas de lograr devoluciones de llamada:

- A través de punteros a funciones (a partir de C# 9)
- A través de delegados

Para ilustrar esto, llamaremos a la siguiente función de Windows en User32.dll, que enumera todos los identificadores de las ventanas de nivel superior:

```
BOOL EnumWindows (WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

WNDENUMPROC es una devolución de llamada que se activa con el identificador de cada ventana en secuencia (o hasta que la devolución de llamada devuelva false). Aquí está su definición:

```
BOOL CALLBACK EnumWindowsProc (HWND hwnd, LPARAM lParam);
```

Devoluciones de llamada con punteros a funciones

A partir de C# 9, la opción más simple y eficiente, cuando tu devolución de llamada es un método estático, es utilizar un puntero a función. En el caso de la devolución de llamada WNDENUMPROC, podemos usar el siguiente puntero a función:

```
delegate*<IntPtr, IntPtr, bool>
```

Esto denota una función que acepta dos argumentos IntPtr y devuelve un valor bool. Luego puedes utilizar el operador & para proporcionarle un método estático:

```
using System;
using System.Runtime.InteropServices;

unsafe
{
  EnumWindows (&PrintWindow, IntPtr.Zero);

  [DllImport ("user32.dll")]
  static extern int EnumWindows (
    delegate*<IntPtr, IntPtr, bool> hWnd, IntPtr lParam);

  static bool PrintWindow (IntPtr hWnd, IntPtr lParam)
  {
    Console.WriteLine (hWnd.ToInt64());
    return true;
  }
}
```

Con los punteros a funciones, la devolución de llamada debe ser un método estático (o una función local estática, como en este ejemplo).

UnmanagedCallersOnly

Puedes mejorar el rendimiento aplicando la palabra clave "unmanaged" a la declaración del puntero a función y el atributo "[UnmanagedCallersOnly]" al método de devolución de llamada:

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

unsafe
{
  EnumWindows (&PrintWindow, IntPtr.Zero);

  [DllImport ("user32.dll")]
  static extern int EnumWindows (
    delegate* unmanaged <IntPtr, IntPtr, byte> hWnd, IntPtr lParam);

  [UnmanagedCallersOnly]
  static byte PrintWindow (IntPtr hWnd, IntPtr lParam)
  {
    Console.WriteLine (hWnd.ToInt64());
    return 1;
  }
}
```

Este atributo marca el método PrintWindow para que solo pueda ser llamado desde código no administrado, lo que permite que el tiempo de ejecución tome atajos. Observa que también hemos cambiado el tipo de retorno del método de bool a byte: esto se debe a que los métodos a los que se aplica [UnmanagedCallersOnly] solo pueden usar tipos de valor "blittable" en la firma. Los tipos "blittable" son aquellos que no requieren ninguna lógica especial de marshalling porque se representan de manera idéntica en los mundos administrado y no administrado. Esto incluye los tipos enteros primitivos, float, double y las estructuras que contienen solo tipos "blittable". El tipo char también es "blittable" si forma parte de una estructura con un atributo de diseño de estructura (StructLayout) que especifica CharSet.Unicode:

```
[StructLayout (LayoutKind.Sequential, CharSet=CharSet.Unicode)]
```

Convenciones de llamada no predeterminadas

De forma predeterminada, el compilador asume que la devolución de llamada no administrada sigue la convención de llamada predeterminada de la plataforma. Si esto no es así, puedes especificar explícitamente su convención de llamada a través del parámetro CallConvs del atributo [UnmanagedCallersOnly]:

```
[UnmanagedCallersOnly (CallConvs = new[] { typeof (CallConvStdcall) })]
static byte PrintWindow (IntPtr hWnd, IntPtr lParam) ...
```

También debes actualizar el tipo de puntero a función insertando un modificador especial después de la palabra clave "unmanaged":

```
delegate* unmanaged[Stdcall] <IntPtr, IntPtr, byte> hWnd, IntPtr lParam);
```



El compilador te permite poner cualquier identificador (como XYZ) dentro de los corchetes, siempre y cuando haya un tipo .NET llamado CallConvXYZ (que sea comprendido por el tiempo de ejecución y coincida con lo que especificaste al aplicar el atributo

[UnmanagedCallersOnly]). Esto facilita que Microsoft agregue nuevas convenciones de llamada en el futuro.

En este caso, hemos especificado StdCall, que es la convención de llamada predeterminada para Windows (Cdecl es la predeterminada para Linux x86). Estas son todas las opciones que se admiten actualmente:

Name	unmanaged modifier	Supporting type
Stdcall	unmanaged[Stdcall]	CallConvStdcall
Cded	unmanaged[Cdecl]	CallConvCdecl
ThisCall	unmanaged[Thiscall]	CallConvThiscall

Devuelve llamadas con Delegados

Las devoluciones de llamada no administradas también se pueden lograr con delegados. Este enfoque funciona en todas las versiones de C# y permite devoluciones de llamada que hacen referencia a métodos de instancia.

Para continuar, primero declara un tipo de delegado con una firma que coincida con la devolución de llamada. Luego puedes pasar una instancia de delegado al método externo:

```
class CallbackFun
{
  delegate bool EnumWindowsCallback (IntPtr hWnd, IntPtr lParam);

  [DllImport("user32.dll")]
  static extern int EnumWindows (EnumWindowsCallback hWnd, IntPtr lParam);

  static bool PrintWindow (IntPtr hWnd, IntPtr lParam)
  {
    Console.WriteLine (hWnd.ToInt64());
    return true;
  }
  static readonly EnumWindowsCallback printWindowFunc = PrintWindow;

  static void Main() => EnumWindows (printWindowFunc, IntPtr.Zero);
}
```

El uso de delegados para devoluciones de llamada no administradas es irónicamente inseguro, ya que es fácil caer en la trampa de permitir que ocurra una devolución de llamada después de que la instancia del delegado queda fuera de alcance (en ese momento, el delegado se vuelve elegible para la recolección de basura). Esto puede resultar en el peor tipo de excepción en tiempo de ejecución, sin una traza de pila útil.

En el caso de devoluciones de llamada de métodos estáticos, puedes evitar esto asignando la instancia del delegado a un campo estático de solo lectura (como lo hicimos en este ejemplo). Con devoluciones de llamada de métodos de instancia, este patrón no ayudará, por lo que debes codificar cuidadosamente para asegurarte de mantener al menos una referencia a la instancia del delegado durante la duración de cualquier posible devolución de llamada. Incluso en ese caso, si hay un error en el lado no administrado, por el cual invoca una devolución de llamada después de haber indicado que no lo haga, es posible que aún deba lidiar con una excepción no

rastreable. Una solución alternativa es definir un tipo de delegado único por función no administrada: esto ayuda en el diagnóstico porque el tipo de delegado se informa en la excepción.

Puedes cambiar la convención de llamada de la devolución de llamada desde la predeterminada de la plataforma aplicando el atributo [UnmanagedFunctionPointer] al delegado:

```
[UnmanagedFunctionPointer (CallingConvention.Cdecl)]
delegate void MyCallback (int foo, short bar);
```

Simulando una Unión en C#

Cada campo en una estructura tiene suficiente espacio para almacenar sus datos. Considera una estructura que contiene un entero y un carácter. Es probable que el entero comience en un desplazamiento de 0 y esté garantizado al menos de cuatro bytes. Por lo tanto, el carácter comenzaría en un desplazamiento de al menos 4. Si por alguna razón, el carácter comenzara en un desplazamiento de 2, cambiarías el valor del entero si asignaras un valor al carácter. Suena caótico, ¿verdad? Curiosamente, el lenguaje C admite una variación de una estructura llamada unión que hace exactamente esto.

En C#, puedes simular esto utilizando `LayoutKind.Explicit` y el atributo `FieldOffset`. Puede resultar desafiante pensar en un caso en el que esto sea útil. Sin embargo, supongamos que deseas reproducir una nota en un sintetizador externo. La API multimedia de Windows proporciona una función para hacer esto mediante el protocolo MIDI:

```
[DllImport ("winmm.dll")]
public static extern uint midiOutShortMsg (IntPtr handle, uint message);
```

El segundo argumento, `message`, describe qué nota se va a reproducir. El problema radica en la construcción de este entero sin signo de 32 bits: se divide internamente en bytes que representan un canal MIDI, una nota y la velocidad a la que se toca. Una solución es realizar desplazamientos y máscaras mediante los operadores `<<`, `>>`, `&` y `|` para convertir estos bytes hacia y desde el mensaje "empaquetado" de 32 bits. Sin embargo, una forma mucho más sencilla es definir una estructura con diseño explícito:

```
[StructLayout (LayoutKind.Explicit)]
public struct NoteMessage
{
  [FieldOffset(0)] public uint PackedMsg;    // 4 bytes long

  [FieldOffset(0)] public byte Channel;      // FieldOffset also at 0
  [FieldOffset(1)] public byte Note;
  [FieldOffset(2)] public byte Velocity;
}
```

Los campos Channel, Note y Velocity se superponen intencionalmente con el mensaje "empaquetado" de 32 bits. Esto te permite leer y escribir utilizando cualquiera de ellos. No se requieren cálculos adicionales para mantener sincronizados los demás campos:

```
NoteMessage n = new NoteMessage();
Console.WriteLine (n.PackedMsg);    // 0
```

```
n.Channel = 10;
n.Note = 100;
n.Velocity = 50;
Console.WriteLine (n.PackedMsg);    // 3302410

n.PackedMsg = 3328010;
Console.WriteLine (n.Note);         // 200
```

Memoria compartida (Shared Memory)

Los archivos mapeados en memoria, o memoria compartida, es una característica en Windows que permite que varios procesos en la misma computadora compartan datos. La memoria compartida es extremadamente rápida y, a diferencia de las tuberías, ofrece acceso aleatorio a los datos compartidos. Vimos en el Capítulo 15 cómo puedes usar la clase MemoryMappedFile para acceder a archivos mapeados en memoria; omitir esto y llamar directamente a los métodos Win32 es una buena manera de demostrar el uso de P/Invoke.

La función Win32 CreateFileMapping asigna memoria compartida. Le indicas cuántos bytes necesitas y el nombre con el que identificarás la compartición. Otra aplicación puede suscribirse a esta memoria llamando a OpenFileMapping con el mismo nombre. Ambos métodos devuelven un identificador, que puedes convertir en un puntero llamando a MapViewOfFile.

Aquí hay una clase que encapsula el acceso a la memoria compartida:

```
using System;
using System.Runtime.InteropServices;
using System.ComponentModel;

public sealed class SharedMem : IDisposable
{
  // Here we're using enums because they're safer than constants

  enum FileProtection : uint      // constants from winnt.h
  {
    ReadOnly = 2,
    ReadWrite = 4
  }

  enum FileRights : uint          // constants from WinBASE.h
  {
    Read = 4,
    Write = 2,
    ReadWrite = Read + Write
  }

  static readonly IntPtr NoFileHandle = new IntPtr (-1);

  [DllImport ("kernel32.dll", SetLastError = true)]
  static extern IntPtr CreateFileMapping (IntPtr hFile,
                                          int lpAttributes,
                                          FileProtection flProtect,
```

```
                                        uint dwMaximumSizeHigh,
                                        uint dwMaximumSizeLow,
                                        string lpName);

  [DllImport ("kernel32.dll", SetLastError=true)]
  static extern IntPtr OpenFileMapping (FileRights dwDesiredAccess,
                                        bool bInheritHandle,
                                        string lpName);

  [DllImport ("kernel32.dll", SetLastError = true)]
  static extern IntPtr MapViewOfFile (IntPtr hFileMappingObject,
                                      FileRights dwDesiredAccess,
                                      uint dwFileOffsetHigh,
                                      uint dwFileOffsetLow,
                                      uint dwNumberOfBytesToMap);

  [DllImport ("Kernel32.dll", SetLastError = true)]
  static extern bool UnmapViewOfFile (IntPtr map);

  [DllImport ("kernel32.dll", SetLastError = true)]
  static extern int CloseHandle (IntPtr hObject);

  IntPtr fileHandle, fileMap;

  public IntPtr Root => fileMap;

  public SharedMem (string name, bool existing, uint sizeInBytes)
  {
    if (existing)
      fileHandle = OpenFileMapping (FileRights.ReadWrite, false, name);
    else
      fileHandle = CreateFileMapping (NoFileHandle, 0,
                                      FileProtection.ReadWrite,
                                      0, sizeInBytes, name);
    if (fileHandle == IntPtr.Zero)
      throw new Win32Exception();

    // Obtain a read/write map for the entire file
    fileMap = MapViewOfFile (fileHandle, FileRights.ReadWrite, 0, 0, 0);

    if (fileMap == IntPtr.Zero)
      throw new Win32Exception();
  }

  public void Dispose()
  {
    if (fileMap != IntPtr.Zero) UnmapViewOfFile (fileMap);
    if (fileHandle != IntPtr.Zero) CloseHandle (fileHandle);
    fileMap = fileHandle = IntPtr.Zero;
  }
}
```

En este ejemplo, establecemos SetLastError=true en los métodos DllImport que utilizan el protocolo SetLastError para emitir códigos de error. Esto asegura que la excepción Win32Exception se rellene con detalles del error cuando se lance esa excepción. (También te permite consultar el error explícitamente llamando a Marshal.GetLastWin32Error).

Para demostrar esta clase, necesitamos ejecutar dos aplicaciones. La primera crea la memoria compartida de la siguiente manera:

```
using (SharedMem sm = new SharedMem ("MyShare", false, 1000))
{
  IntPtr root = sm.Root;
  // I have shared memory!

  Console.ReadLine();         // Here's where we start a second app...
}
```

La segunda aplicación se suscribe a la memoria compartida construyendo un objeto SharedMem con el mismo nombre y el argumento `true`:

```
using (SharedMem sm = new SharedMem ("MyShare", true, 1000))
{
  IntPtr root = sm.Root;
  // I have the same shared memory!
  // ...
}
```

El resultado neto es que cada programa tiene un IntPtr, un puntero a la misma memoria no administrada. Ahora, las dos aplicaciones necesitan leer y escribir en la memoria a través de este puntero común. Un enfoque es escribir una clase que encapsule todos los datos compartidos y luego serializar (y deserializar) los datos en la memoria no administrada utilizando un UnmanagedMemoryStream. Sin embargo, esto es ineficiente si hay mucha cantidad de datos. Imagina si la clase de memoria compartida tuviera un megabyte de datos y solo se necesita actualizar un entero. Un enfoque mejor es definir la estructura de datos compartidos como una struct y luego mapearla directamente en la memoria compartida. Esto lo discutiremos en la siguiente sección.

Mapear una struct a memoria no administrada

Puedes mapear directamente una struct con StructLayout Sequential o Explicit a memoria no administrada. Considera la siguiente struct:

```
[StructLayout (LayoutKind.Sequential)]
unsafe struct MySharedData
{
  public int Value;
  public char Letter;
  public fixed float Numbers [50];
}
```

La directiva "fixed" nos permite definir matrices de tipo de valor de longitud fija en línea, y es lo que nos lleva al ámbito inseguro. El espacio en esta estructura se asigna en línea para 50 números de punto flotante. A diferencia de las matrices estándar de C#, "Numbers" no es una referencia a una matriz, es la matriz en sí. Si ejecutamos lo siguiente:

```
static unsafe void Main() => Console.WriteLine (sizeof (MySharedData));
```

el resultado es 208: 50 floats de cuatro bytes, más los cuatro bytes para el entero "Value", más dos bytes para el carácter "Letter". El total, 206, se redondea a 208 debido a que los floats se alinean en límites de cuatro bytes (cuatro bytes es el tamaño de un float).

Podemos demostrar MySharedData en un contexto inseguro de la forma más sencilla con memoria asignada en la pila ("stackalloc"):

```
MySharedData d;
MySharedData* data = &d;       // Get the address of d

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

o:

```
// Allocate the array on the stack:
MySharedData* data = stackalloc MySharedData[1];

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

Por supuesto, no estamos demostrando nada que no se pueda lograr en un contexto administrado. Sin embargo, supongamos que queremos almacenar una instancia de MySharedData en el montón no administrado, fuera del ámbito del recolector de basura de CLR. Aquí es donde los punteros se vuelven realmente útiles:

```
MySharedData* data = (MySharedData*)
  Marshal.AllocHGlobal (sizeof (MySharedData)).ToPointer();

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

Marshal.AllocHGlobal asigna memoria en el montón no administrado. Aquí te muestro cómo liberar la misma memoria más tarde:

```
Marshal.FreeHGlobal (new IntPtr (data));
```

(El olvido de liberar la memoria resulta en una fuga de memoria de la vieja escuela).



A partir de .NET 6, en su lugar puedes usar la nueva clase NativeMemory para asignar y liberar memoria no administrada. NativeMemory utiliza una API subyacente más nueva (y mejor) que AllocHGlobal y también incluye métodos para realizar asignaciones alineadas.

Siguiendo su nombre, aquí usamos MySharedData en conjunto con la clase SharedMem que escribimos en la sección anterior. El siguiente programa asigna un bloque de memoria compartida y luego mapea la estructura MySharedData en esa memoria:

```
static unsafe void Main()
{
  using (SharedMem sm = new SharedMem ("MyShare", false,
                          (uint) sizeof (MySharedData)))
  {
    void* root = sm.Root.ToPointer();
    MySharedData* data = (MySharedData*) root;

    data->Value = 123;
    data->Letter = 'X';
    data->Numbers[10] = 1.45f;
    Console.WriteLine ("Written to shared memory");

    Console.ReadLine();

    Console.WriteLine ("Value is " + data->Value);
    Console.WriteLine ("Letter is " + data->Letter);
    Console.WriteLine ("11th Number is " + data->Numbers[10]);
    Console.ReadLine();
  }
}
```



Puedes usar la clase integrada MemoryMappedFile en lugar de SharedMem, de la siguiente manera:

```
using (MemoryMappedFile mmFile =
       MemoryMappedFile.CreateNew ("MyShare", 1000))
using (MemoryMappedViewAccessor accessor =
       mmFile.CreateViewAccessor())
{
  byte* pointer = null;
  accessor.SafeMemoryMappedViewHandle.AcquirePointer
   (ref pointer);
  void* root = pointer;
  ...
}
```

Aquí hay un segundo programa que se conecta a la misma memoria compartida, leyendo los valores escritos por el primer programa (debe ejecutarse mientras el primer programa está esperando en la instrucción ReadLine porque el objeto de memoria compartida se elimina al salir de su declaración using):

```
static unsafe void Main()
{
  using (SharedMem sm = new SharedMem ("MyShare", true,
                          (uint) sizeof (MySharedData)))
  {
    void* root = sm.Root.ToPointer();
    MySharedData* data = (MySharedData*) root;

    Console.WriteLine ("Value is " + data->Value);
    Console.WriteLine ("Letter is " + data->Letter);
    Console.WriteLine ("11th Number is " + data->Numbers[10]);

    // Our turn to update values in shared memory!
    data->Value++;
    data->Letter = '!';
```

```
        data->Numbers[10] = 987.5f;
        Console.WriteLine ("Updated shared memory");
        Console.ReadLine();
      }
    }
```

El resultado de cada uno de estos programas es el siguiente:

```
// First program:

Written to shared memory
Value is 124
Letter is !
11th Number is 987.5

// Second program:

Value is 123
Letter is X
11th Number is 1.45
Updated shared memory
```

No te desanimes por el uso de punteros: los programadores de C++ los utilizan en aplicaciones completas y logran que todo funcione. ¡Al menos la mayoría de las veces! Este tipo de uso es bastante simple en comparación.

Como sucede, nuestro ejemplo es inseguro, literalmente, por otra razón. No hemos considerado los problemas de seguridad de subproceso (o más precisamente, de seguridad de proceso) que surgen cuando dos programas acceden a la misma memoria al mismo tiempo. Para usar esto en una aplicación de producción, necesitaríamos agregar la palabra clave volatile a los campos Value y Letter en la estructura MySharedData para evitar que los campos sean almacenados en caché por el compilador Just-in-Time (JIT) (o por el hardware en los registros de la CPU). Además, a medida que nuestra interacción con los campos creciera más allá de lo trivial, es probable que necesitaríamos proteger su acceso mediante un Mutex entre procesos, tal como usaríamos declaraciones lock para proteger el acceso a los campos en un programa con múltiples subprocesos. Discutimos la seguridad de subprocesos en detalle en el Capítulo 21.

fixed y fixed {...}

Una limitación de mapear estructuras directamente en la memoria es que la estructura solo puede contener tipos no administrados. Si necesitas compartir datos de cadena, por ejemplo, debes usar una matriz de caracteres fijos en su lugar. Esto significa realizar conversiones manuales hacia y desde el tipo de cadena. Aquí te mostramos cómo hacerlo:

```
[StructLayout (LayoutKind.Sequential)]
unsafe struct MySharedData
{
  ...
  // Allocate space for 200 chars (i.e., 400 bytes).
  const int MessageSize = 200;
  fixed char message [MessageSize];

  // One would most likely put this code into a helper class:
```

```
    public string Message
    {
      get { fixed (char* cp = message) return new string (cp); }
      set
      {
        fixed (char* cp = message)
        {
          int i = 0;
          for (; i < value.Length && i < MessageSize - 1; i++)
            cp [i] = value [i];

          // Add the null terminator
          cp [i] = '\0';
        }
      }
    }
  }
```



No existe tal cosa como una referencia a una matriz fija; en su lugar, obtienes un puntero. Cuando accedes a un índice de una matriz fija, en realidad estás realizando una aritmética de punteros.

En el primer uso de la palabra clave fixed, asignamos espacio, en línea, para 200 caracteres en la estructura. La misma palabra clave (de manera confusa) tiene un significado diferente cuando se utiliza más adelante en la definición de la propiedad. Indica al CLR que ancle un objeto para que, en caso de que decida realizar una recolección de basura dentro del bloque fixed, no mueva la estructura subyacente en el montón de memoria (porque sus contenidos se están iterando mediante punteros de memoria directos). Al observar nuestro programa, podrías preguntarte cómo MySharedData podría cambiar en la memoria, dado que no reside en el montón sino en el mundo no administrado, donde el recolector de basura no tiene jurisdicción. Sin embargo, el compilador no lo sabe y está preocupado de que podríamos usar MySharedData en un contexto administrado, por lo que insiste en que agreguemos la palabra clave fixed para hacer que nuestro código inseguro sea seguro en contextos administrados. Y el compilador tiene razón, esto es todo lo que se necesitaría para poner MySharedData en el montón:

```
object obj = new MySharedData();
```

Esto resulta en un MySharedData encajado (boxed) en el montón y elegible para su movimiento durante la recolección de basura.

Este ejemplo ilustra cómo una cadena puede ser representada en una estructura mapeada a memoria no administrada. Para tipos más complejos, también tienes la opción de usar código de serialización existente. La única condición es que los datos serializados nunca deben exceder, en longitud, su asignación de espacio en la estructura; de lo contrario, el resultado es una unión no intencionada con los campos posteriores.

Interoperabilidad con COM

El tiempo de ejecución de .NET proporciona soporte especial para COM, lo que permite el uso de objetos COM desde .NET y viceversa. COM está disponible solo en Windows.

The Purpose of COM (El propósito de COM)

COM (Component Object Model) es un estándar binario para la interconexión con bibliotecas, lanzado por Microsoft en 1993. La motivación para inventar COM fue permitir que los componentes se comunicaran entre sí de manera independiente del lenguaje y tolerante a versiones. Antes de COM, el enfoque en Windows era publicar DLL que declaraban estructuras y funciones utilizando el lenguaje de programación C. Este enfoque no solo es específico del lenguaje, sino que también es frágil. La especificación de un tipo en dicha biblioteca está inseparable de su implementación: incluso actualizar una estructura con un nuevo campo significa romper su especificación.

La belleza de COM fue separar la especificación de un tipo de su implementación subyacente a través de un concepto conocido como interfaz COM. COM también permitía llamar métodos en objetos con estado, en lugar de limitarse a llamadas de procedimiento simples.



En cierto sentido, el modelo de programación de .NET es una evolución de los principios de programación de COM: la plataforma .NET también facilita el desarrollo interlenguaje y permite que los componentes binarios evolucionen sin romper las aplicaciones que dependen de ellos.

The Basics of the COM Type System (Los fundamentos del sistema de tipos de COM)

Los fundamentos del sistema de tipos de COM se basan en las interfaces. Una interfaz COM es similar a una interfaz .NET, pero es más prevalente porque un tipo COM expone su funcionalidad solo a través de una interfaz. En el mundo .NET, por ejemplo, podríamos declarar un tipo simplemente de la siguiente manera:

```
public class Foo
{
  public string Test() => "Hello, world";
}
```

Los consumidores de ese tipo pueden usar `Foo` directamente. Y si más tarde cambiáramos la implementación de `Test()`, las llamadas de las bibliotecas no requerirían una recompilación. En este sentido, .NET separa la interfaz de la implementación, sin requerir interfaces. Incluso podríamos agregar una sobrecarga sin romper las llamadas existentes:

```
public string Test (string s) => $"Hello, world {s}";
```

En el mundo COM, `Foo` expone su funcionalidad a través de una interfaz para lograr el mismo desacoplamiento. Entonces, en la biblioteca de tipos de `Foo`, existiría una interfaz como esta:

```
public interface IFoo { string Test(); }
```

(La hemos ilustrado mostrando una interfaz de C# en lugar de una interfaz COM. El principio, sin embargo, es el mismo, aunque la implementación subyacente es diferente).

Los llamadores interactuarían con `IFoo` en lugar de `Foo`.

Cuando se trata de agregar la versión sobrecargada de Test, la vida es más complicada con COM que con .NET. Primero, evitaríamos modificar la interfaz IFoo, ya que esto rompería la compatibilidad binaria con la versión anterior (uno de los principios de COM es que las interfaces, una vez publicadas, son inmutables). Segundo, COM no permite la sobrecarga de métodos. La solución es hacer que Foo implemente una segunda interfaz:

```
public interface IFoo2 { string Test (string s); }
```

(Sin embargo, hemos transliterado esto a una interfaz de .NET para mayor familiaridad).

Soportar múltiples interfaces es de vital importancia para hacer que las bibliotecas COM sean versionables.

IUnknown y IDispatch

Todas las interfaces de COM se identifican con un Identificador Único Global (GUID). La interfaz raíz en COM es IUnknown, todos los objetos COM deben implementarla. Esta interfaz tiene tres métodos:

- AddRef
- Release
- QueryInterface

AddRef y Release se utilizan para la gestión del ciclo de vida, dado que COM utiliza recuento de referencias en lugar de recolección automática de basura (COM fue diseñado para trabajar con código no administrado, donde la recolección automática de basura no es factible). El método QueryInterface devuelve una referencia de objeto que admite esa interfaz, si es posible.

Para habilitar la programación dinámica (por ejemplo, scripting y automatización), un objeto COM también puede implementar IDispatch. Esto permite que lenguajes dinámicos como VBScript llamen a objetos COM de manera tardía, de forma similar a "dynamic" en C# (aunque solo para invocaciones simples).

Llamando a un componente COM desde C#

El soporte integrado de CLR para COM significa que no trabajas directamente con IUnknown e IDispatch. En su lugar, trabajas con objetos de CLR y el tiempo de ejecución enmarca tus llamadas al mundo COM a través de Runtime-Callable Wrappers (RCWs). El tiempo de ejecución también maneja la gestión del ciclo de vida llamando a AddRef y Release (cuando el objeto .NET se finaliza) y se encarga de las conversiones de tipo primitivo entre los dos mundos. La conversión de tipos asegura que cada lado vea, por ejemplo, los tipos enteros y cadenas en sus formas familiares.

Además, es necesario tener alguna forma de acceder a los RCW de manera estáticamente tipada. Esto es responsabilidad de los tipos de interoperabilidad COM. Los tipos de interoperabilidad

COM son tipos de proxy generados automáticamente que exponen un miembro .NET por cada miembro COM. La herramienta de importación de bibliotecas de tipos (tlbimp.exe) genera tipos de interoperabilidad COM desde la línea de comandos, basándose en una biblioteca COM que elijas, y los compila en un ensamblado de interoperabilidad COM.



Si un componente COM implementa múltiples interfaces, la herramienta tlbimp.exe genera un solo tipo que contiene una unión de miembros de todas las interfaces.

Puedes crear un ensamblado de interoperabilidad COM en Visual Studio yendo al cuadro de diálogo Agregar referencia y eligiendo una biblioteca de la pestaña COM. Por ejemplo, si tienes Microsoft Excel instalado, al agregar una referencia a la Biblioteca de objetos de Microsoft Excel, puedes interoperar con las clases COM de Excel. Aquí tienes el código en C# para crear y mostrar un libro de trabajo y luego llenar una celda en ese libro de trabajo:

```
using System;
using Excel = Microsoft.Office.Interop.Excel;

var excel = new Excel.Application();
excel.Visible = true;
Excel.Workbook workBook = excel.Workbooks.Add();
((Excel.Range)excel.Cells[1, 1]).Font.FontStyle = "Bold";
((Excel.Range)excel.Cells[1, 1]).Value2 = "Hello World";
workBook.SaveAs (@"d:\temp.xlsx");
```



Actualmente, es necesario incrustar los tipos de interoperabilidad en tu aplicación (de lo contrario, el tiempo de ejecución no los localizará en tiempo de ejecución). Puedes hacerlo haciendo clic en la referencia COM en el Explorador de soluciones de Visual Studio y estableciendo la propiedad "Embed Interop Types" en true en la ventana de propiedades. También puedes abrir el archivo .csproj y agregar la siguiente línea (en negrita):

```
<ItemGroup>
  <COMReference Include="Microsoft.Office.Excel.dll">
    ...
    <EmbedInteropTypes>true</EmbedInteropTypes>
  </COMReference>
</ItemGroup>
```

La clase `Excel.Application` es un tipo de interoperabilidad COM cuyo tipo de tiempo de ejecución es un RCW. Cuando accedemos a las propiedades `Workbooks` y `Cells`, obtenemos más tipos de interoperabilidad.

Parámetros opcionales y argumentos con nombre

Debido a que las API de COM no admiten la sobrecarga de funciones, es muy común tener funciones con numerosos parámetros, muchos de los cuales son opcionales. Por ejemplo, así es como podrías llamar al método `Save` de un libro de trabajo de Excel:

```
var missing = System.Reflection.Missing.Value;

workBook.SaveAs (@"d:\temp.xlsx", missing, missing, missing, missing,
  missing, Excel.XlSaveAsAccessMode.xlNoChange, missing, missing,
  missing, missing, missing);
```

La buena noticia es que el soporte de C# para parámetros opcionales es compatible con COM, por lo que simplemente podemos hacer lo siguiente:

```
workBook.SaveAs (@"d:\temp.xlsx");
```

(Como mencionamos en el Capítulo 3, los parámetros opcionales son "expandidos" por el compilador en su forma completa y detallada).

Los argumentos con nombre te permiten especificar argumentos adicionales, independientemente de su posición:

```
workBook.SaveAs (@"c:\test.xlsx", Password:"foo");
```

Parámetros implícitos `ref`

Algunas API de COM (especialmente Microsoft Word) exponen funciones que declaran cada parámetro como paso por referencia, sin importar si la función modifica o no el valor del parámetro. Esto se debe a la supuesta mejora de rendimiento al evitar la copia de los valores de los argumentos (la mejora real de rendimiento es insignificante).

Históricamente, llamar a tales métodos desde C# ha sido incómodo porque debes especificar la palabra clave `ref` con cada argumento, lo que impide el uso de parámetros opcionales. Por ejemplo, para abrir un documento de Word, solíamos hacer lo siguiente:

```
object filename = "foo.doc";
object notUsed1 = Missing.Value;
object notUsed2 = Missing.Value;
object notUsed3 = Missing.Value;
...
Open (ref filename, ref notUsed1, ref notUsed2, ref notUsed3, ...);
```

Gracias a los parámetros implícitos `ref`, puedes omitir el modificador `ref` en las llamadas a funciones COM, lo que permite el uso de parámetros opcionales:

```
word.Open ("foo.doc");
```

La advertencia es que no obtendrás errores en tiempo de compilación ni en tiempo de ejecución si el método COM que estás llamando realmente modifica un valor de argumento.

Indexadores

La capacidad de omitir el modificador `ref` tiene otro beneficio: permite acceder a indexadores COM con parámetros `ref` mediante la sintaxis de indexadores ordinarios de C#. De lo contrario,

esto estaría prohibido porque los parámetros `ref/out` no son compatibles con los indexadores de C#.

También puedes llamar a propiedades COM que aceptan argumentos. En el siguiente ejemplo, `Foo` es una propiedad que acepta un argumento entero:

```
myComObject.Foo [123] = "Hello";
```

Aún está prohibido escribir este tipo de propiedades en C#: un tipo solo puede exponer un indexador en sí mismo (el indexador "predeterminado"). Por lo tanto, si quisieras escribir código en C# que haga que la declaración anterior sea legal, `Foo` debería devolver otro tipo que exponga un indexador (predeterminado).

Enlace Dinámico

Hay dos formas en las que el enlace dinámico puede ayudar al llamar a componentes COM.

La primera forma es permitiendo el acceso a un componente COM sin un tipo de interoperabilidad COM. Para hacer esto, llama a `Type.GetTypeFromProgID` con el nombre del componente COM para obtener una instancia COM, y luego usa el enlace dinámico para llamar a los miembros a partir de ahí. Por supuesto, no hay IntelliSense y no se pueden realizar comprobaciones en tiempo de compilación:

```
Type excelAppType = Type.GetTypeFromProgID ("Excel.Application", true);
dynamic excel = Activator.CreateInstance (excelAppType);
excel.Visible = true;
dynamic wb = excel.Workbooks.Add();
excel.Cells [1, 1].Value2 = "foo";
```

(Lo mismo se puede lograr, de manera mucho más torpe, con reflexión en lugar de enlace dinámico).



Una variación de este enfoque es llamar a un componente COM que solo admite `IDispatch`. Sin embargo, estos componentes son bastante raros.

El enlace dinámico también puede ser útil (en menor medida) al tratar con el tipo de variant de COM. Por razones más relacionadas con un diseño deficiente que por necesidad, las funciones de la API de COM a menudo están llenas de este tipo, que es aproximadamente equivalente a `object` en .NET. Si habilitas "Embed Interop Types" en tu proyecto (más sobre esto pronto), el tiempo de ejecución mapeará `variant` a `dynamic`, en lugar de mapearlo a `object`, evitando la necesidad de realizar conversiones. Por ejemplo, podrías hacer legítimamente:

```
excel.Cells [1, 1].Font.FontStyle = "Bold";
```

en lugar de:

```
var range = (Excel.Range) excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
```

La desventaja de trabajar de esta manera es que pierdes el autocompletado, por lo que debes saber que existe una propiedad llamada `Font`. Por esta razón, generalmente es más fácil asignar dinámicamente el resultado a su tipo de interoperabilidad conocido:

```
Excel.Range range = excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
```

Como puedes ver, ¡esto ahorra solo cinco caracteres en comparación con el enfoque anticuado!

El mapeo de `variant` a `dynamic` es el valor predeterminado y es una función de habilitar "Embed Interop Types" en una referencia.

Incrustación de Tipos de Interoperabilidad (Embedding Interop Types)

Mencionamos anteriormente que C# por lo general llama a componentes COM a través de tipos de interoperabilidad generados mediante la herramienta `tlbimp.exe` (directamente o a través de Visual Studio).

Históricamente, tu única opción era referenciar ensamblados de interoperabilidad de la misma manera que lo harías con cualquier otro ensamblado. Esto podía ser problemático porque los ensamblados de interoperabilidad pueden volverse bastante grandes con componentes COM complejos. Por ejemplo, un complemento muy pequeño para Microsoft Word requiere un ensamblado de interoperabilidad que es varias veces más grande que él mismo.

En lugar de hacer referencia a un ensamblado de interoperabilidad, tienes la opción de incrustar las porciones que utilizas. El compilador analiza el ensamblado para determinar exactamente los tipos y miembros que tu aplicación requiere, e incrusta definiciones solo para esos tipos y miembros directamente en tu aplicación. Esto evita el exceso de tamaño, así como la necesidad de enviar un archivo adicional.

Para habilitar esta función, selecciona la referencia a COM en el Explorador de soluciones de Visual Studio y establece "Embed Interop Types" en true en la ventana de propiedades, o edita tu archivo .csproj como describimos anteriormente (ver "Calling a COM Component from C#" en la página 979).

Equivalencia de Tipos

El CLR admite la equivalencia de tipos para los tipos de interoperabilidad vinculados. Esto significa que si dos ensamblados enlazan a un mismo tipo de interoperabilidad, esos tipos se considerarán equivalentes si encapsulan el mismo tipo COM. Esto sigue siendo cierto incluso si los ensamblados de interoperabilidad a los que enlazaron fueron generados de forma independiente.

La equivalencia de tipos se basa en el atributo TypeIdentifierAttribute en el espacio de nombres System.Runtime.InteropServices. El compilador aplica automáticamente este atributo cuando enlazas a ensamblados de interoperabilidad. Los tipos COM se consideran equivalentes si tienen el mismo GUID.

Exponiendo Objetos de C# a COM

También es posible escribir clases en C# que pueden ser consumidas en el mundo COM. El CLR lo hace posible a través de un proxy llamado Envoltura Llamable por COM (CCW por sus siglas en inglés). Un CCW realiza el marshaling de tipos entre los dos mundos (como con un RCW) e implementa IUnknown (y opcionalmente IDispatch) según lo requiera el protocolo COM. Un CCW tiene su tiempo de vida controlado desde el lado COM mediante el recuento de referencias (en lugar del recolector de basura de CLR).

Puedes exponer cualquier clase pública a COM (como un "servidor en proceso"). Para hacerlo, primero crea una interfaz, asígnale un GUID único (en Visual Studio, puedes usar la herramienta "Create GUID" en Tools), declara que es visible para COM y luego establece el tipo de interfaz:

```
namespace MyCom
{
  [ComVisible(true)]
  [Guid ("226E5561-C68E-4B2B-BD28-25103ABCA3B1")]  // Change this GUID
  [InterfaceType (ComInterfaceType.InterfaceIsIUnknown)]
  public interface IServer
  {

    int Fibonacci();
  }
}
```

A continuación, proporciona una implementación de tu interfaz, asignando un GUID único a esa implementación:

```
namespace MyCom
{
  [ComVisible(true)]
  [Guid ("09E01FCD-9970-4DB3-B537-0EC555967DD9")]  // Change this GUID
  public class Server
  {
    public ulong Fibonacci (ulong whichTerm)
    {
      if (whichTerm < 1) throw new ArgumentException ("...");
      ulong a = 0;
      ulong b = 1;
      for (ulong i = 0; i < whichTerm; i++)
      {
        ulong tmp = a;
        a = b;
        b = tmp + b;
      }
      return a;
    }
  }
}
```

Edita tu archivo .csproj, agregando la siguiente línea (en negrita):

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.0</TargetFramework>
  <EnableComHosting>true</EnableComHosting>
</PropertyGroup>
```

Ahora, cuando compilas tu proyecto, se generará un archivo adicional llamado MyCom.comhost.dll, que se puede registrar para la interoperabilidad COM. (Recuerda que el archivo siempre será de 32 o 64 bits según la configuración de tu proyecto: no existe la opción

"Any CPU" en este escenario). Desde un símbolo del sistema elevado, cambia al directorio que contiene tu DLL y ejecuta el comando `regsvr32 MyCom.comhost.dll`. Luego podrás consumir tu componente COM desde la mayoría de los lenguajes compatibles con COM.

Por ejemplo, puedes crear este Script de Visual Basic en un editor de texto y ejecutarlo haciendo doble clic en el archivo en el Explorador de Windows o iniciándolo desde un símbolo del sistema como lo harías con cualquier programa:

```
REM Save file as ComClient.vbs
Dim obj
Set obj = CreateObject("MyCom.Server")

result = obj.Fibonacci(12)
Wscript.Echo result
```

Ten en cuenta que el .NET Framework no se puede cargar en el mismo proceso que .NET 5+ o .NET Core. Por lo tanto, un servidor COM de .NET 5+ no se puede cargar en un proceso de cliente COM de .NET Framework, y viceversa.

Enabling Registry-Free COM (Habilitando COM sin registro)

Para habilitar el registro COM sin utilizar el registro del sistema, se utiliza el registro COM sin registro, que utiliza un archivo de manifiesto en lugar del registro para controlar la activación del objeto. Para habilitar esta función, agrega la siguiente línea (en negrita) a tu archivo .csproj:

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.0</TargetFramework>
  <EnableComHosting>true</EnableComHosting>
  <EnableRegFreeCom>true</EnableRegFreeCom>
</PropertyGroup>
```

Luego, al compilar, se generará un archivo MyCom.X.manifest.



En .NET 5+ no hay soporte para generar una biblioteca de tipos COM (*.tlb) automáticamente. En su lugar, puedes escribir manualmente un archivo IDL (Interface Definition Language) o un encabezado de C++ para las declaraciones nativas en tu interfaz.

El lenguaje de expresiones regulares identifica patrones de caracteres. Los tipos de .NET que admiten expresiones regulares se basan en las expresiones regulares de Perl 5 y admiten tanto la funcionalidad de búsqueda como la de búsqueda/reemplazo.

Las expresiones regulares se utilizan para tareas como:

- Validar la entrada de texto, como contraseñas y números de teléfono.
- Analizar datos textuales en formas más estructuradas (por ejemplo, una cadena de versión de NuGet).
- Reemplazar patrones de texto en un documento (por ejemplo, solo palabras completas).

Este capítulo se divide en secciones conceptuales que enseñan los conceptos básicos de las expresiones regulares en .NET, y secciones de referencia que describen el lenguaje de expresiones regulares.

Todos los tipos de expresiones regulares se definen en System.Text.RegularExpressions.



Los ejemplos en este capítulo se cargan previamente en LINQPad, que también incluye una herramienta interactiva de expresiones regulares (presiona Ctrl+Shift+F1). También hay una herramienta en línea disponible en http://regexstorm.net/tester.

Conceptos básicos de las expresiones regulares

Uno de los operadores de expresiones regulares más comunes es un cuantificador. "?" es un cuantificador que coincide con el elemento anterior 0 o 1 vez. En otras palabras, "?" significa opcional. Un elemento puede ser un solo carácter o una estructura compleja de caracteres entre corchetes. Por ejemplo, la expresión regular "colou?r" coincide con color y colour, pero no con colouur.

```
Console.WriteLine (Regex.Match ("color",   @"colou?r").Success);  // True
Console.WriteLine (Regex.Match ("colour",  @"colou?r").Success);  // True
Console.WriteLine (Regex.Match ("colouur", @"colou?r").Success);  // False
```

La función Regex.Match busca dentro de una cadena más grande. El objeto que devuelve tiene propiedades para el índice y la longitud de la coincidencia, así como el valor real que coincide:

```
Match m = Regex.Match ("any colour you like", @"colou?r");

Console.WriteLine (m.Success);     // True
Console.WriteLine (m.Index);       // 4
Console.WriteLine (m.Length);      // 6
Console.WriteLine (m.Value);       // colour
Console.WriteLine (m.ToString());  // colour
```

Puedes pensar en Regex.Match como una versión más potente del método IndexOf de la cadena. La diferencia es que busca un patrón en lugar de una cadena literal.

El método IsMatch es un atajo para llamar a Match y luego probar la propiedad Success.

El motor de expresiones regulares funciona de izquierda a derecha de forma predeterminada, por lo que solo se devuelve la coincidencia más a la izquierda. Puedes usar el método NextMatch para obtener más coincidencias:

```
Match m1 = Regex.Match ("One color? There are two colours in my head!",
                        @"colou?rs?");
Match m2 = m1.NextMatch();
Console.WriteLine (m1);          // color
Console.WriteLine (m2);          // colours
```

El método Matches devuelve todas las coincidencias en un arreglo. Podemos reescribir el ejemplo anterior de la siguiente manera:

```
foreach (Match m in Regex.Matches
          ("One color? There are two colours in my head!", @"colou?rs?"))
  Console.WriteLine (m);
```

Otro operador común de expresiones regulares es el alternador, representado por una barra vertical, |. Un alternador expresa alternativas. Lo siguiente coincide con "Jen", "Jenny" y "Jennifer":

```
Console.WriteLine (Regex.IsMatch ("Jenny", "Jen(ny|nifer)?"));  // True
```

Los corchetes alrededor de un alternador separan las alternativas del resto de la expresión.



Puedes especificar un tiempo de espera al realizar coincidencias de expresiones regulares. Si una operación de coincidencia tarda más que el TimeSpan especificado, se lanzará una excepción RegexMatchTimeoutException. Esto puede ser útil si tu programa procesa expresiones regulares arbitrarias (por ejemplo, en un cuadro de diálogo de búsqueda avanzada) porque evita que las expresiones regulares mal formadas giren infinitamente.

Expresiones regulares compiladas

En algunos de los ejemplos anteriores, llamamos repetidamente a un método RegEx estático con el mismo patrón. Un enfoque alternativo en estos casos es instanciar un objeto Regex con el patrón y RegexOptions.Compiled y luego llamar a los métodos de instancia:

```
Regex r = new Regex (@"sausages?", RegexOptions.Compiled);
Console.WriteLine (r.Match ("sausage"));   // sausage
Console.WriteLine (r.Match ("sausages"));  // sausages
```

RegexOptions.Compiled instruye a la instancia de RegEx a utilizar generación de código ligero (DynamicMethod en Reflection.Emit) para construir y compilar dinámicamente un código adaptado a esa expresión regular en particular. Esto resulta en una búsqueda más rápida, a costa de un costo de compilación inicial.

También puedes instanciar un objeto Regex sin utilizar RegexOptions.Compiled. Una instancia de Regex es inmutable.



El motor de expresiones regulares es rápido. Incluso sin compilación, una coincidencia simple generalmente tarda menos de un microsegundo.

RegexOptions

El enum RegexOptions te permite ajustar el comportamiento de la coincidencia. Un uso común de RegexOptions es realizar una búsqueda que no distinga entre mayúsculas y minúsculas:

```
Console.WriteLine (Regex.Match ("a", "A", RegexOptions.IgnoreCase)); // a
```

Esto aplica las reglas de equivalencia de mayúsculas y minúsculas de la cultura actual. La bandera CultureInvariant te permite solicitar la cultura invariante en su lugar:

```
Console.WriteLine (Regex.Match ("a", "A", RegexOptions.IgnoreCase
                                        | RegexOptions.CultureInvariant));
```

Puedes activar la mayoría de las banderas de RegexOptions dentro de una expresión regular en sí misma, utilizando un código de una sola letra, de la siguiente manera:

```
Console.WriteLine (Regex.Match ("a", @"(?i)A"));                    // a
```

Puedes activar y desactivar opciones en toda una expresión:

```
Console.WriteLine (Regex.Match ("AAAa", @"(?i)a(?-i)a"));           // Aa
```

Otra opción útil es IgnorePatternWhitespace o (?x). Esto te permite insertar espacios en blanco para que una expresión regular sea más legible, sin que el espacio en blanco se tome literalmente.

La tabla 25-1 enumera todos los valores de RegExOptions junto con sus códigos de una sola letra.

Table 25-1. Regular expression options

Enum value	Regular expressions code	Description
None		
IgnoreCase	i	Ignores case (by default, regular expressions are case sensitive)
Multiline	m	Changes ^ and $ so that they match the start/end of a line instead of start/end of the string
ExplicitCapture	n	Captures only explicitly named or explicitly numbered groups (see "Groups" on page 996)
Compiled		Forces compilation to IL (see "Compiled Regular Expressions" on page 989)
Singleline	s	Makes . match every character (instead of matching every character except \n)
IgnorePatternWhitespace	x	Eliminates unescaped whitespace from the pattern
RightToLeft	r	Searches from right to left; can't be specified midstream
ECMAScript		Forces ECMA compliance (by default, the implementation is not ECMA compliant)
CultureInvariant		Turns off culture-specific behavior for string comparisons

Caracteres de escape

Las expresiones regulares tienen los siguientes metacaracteres, que tienen un significado especial en lugar de literal:

```
\ * + ? | { [ () ^ $ . #
```

Para usar un metacaracter literalmente, debes precederlo, o escaparlo, con una barra invertida. En el siguiente ejemplo, escapamos el carácter ? para que coincida con la cadena "what?":

```
Console.WriteLine (Regex.Match ("what?", @"what\?")); // what? (correct)
Console.WriteLine (Regex.Match ("what?", @"what?"));  // what  (incorrect)
```



Si el carácter está dentro de un conjunto (corchetes), esta regla no se aplica y los metacaracteres se interpretan literalmente. Discutiremos los conjuntos en la siguiente sección.

Los métodos Escape y Unescape de Regex convierten una cadena que contiene metacaracteres de expresiones regulares reemplazándolos con sus equivalentes escapados, y viceversa:

```
Console.WriteLine (Regex.Escape   (@"?"));     // \?
Console.WriteLine (Regex.Unescape (@"\?"));    // ?>
```

Todas las cadenas de expresiones regulares en este capítulo se expresan con el literal @ de C#. Esto es para evitar el mecanismo de escape de C#, que también utiliza la barra invertida. Sin el @, una barra invertida literal requeriría cuatro barras invertidas:

```
Console.WriteLine (Regex.Match ("\\", "\\\\"));    // \
```

A menos que incluyas la opción (?x), los espacios se tratan literalmente en las expresiones regulares:

```
Console.Write (Regex.IsMatch ("hello world", @"hello world"));  // True
```

Conjuntos de caracteres

Los conjuntos de caracteres actúan como comodines para un conjunto particular de caracteres.

Expression	Meaning	Inverse ("not")
`[abcdef]`	Matches a single character in the list.	`[^abcdef]`
`[a-f]`	Matches a single character in a *range*.	`[^a-f]`
`\d`	Matches anything in the Unicode *digits* category. In ECMAScript mode, `[0-9]`.	`\D`
`\w`	Matches a *word* character (by default, varies according to `CultureInfo.CurrentCulture`; for example, in English, same as `[a-zA-Z_0-9]`).	`\W`
`\s`	Matches a whitespace character; that is, anything for which `char.IsWhiteSpace` returns true (including Unicode spaces). In ECMAScript mode, `[\n\r\t\f\v ]`.	`\S`
`\p{`*category*`}`	Matches a character in a specified *category*.	`\P`
`.`	(Default mode) Matches any character except `\n`.	`\n`
`.`	(`SingleLine` mode) Matches any character.	`\n`

Para hacer coincidir exactamente uno de un conjunto de caracteres, coloca el conjunto de caracteres entre corchetes:

```
Console.Write (Regex.Matches ("That is that.", "[Tt]hat").Count);   // 2
```

Para hacer coincidir cualquier caracter excepto los que están en un conjunto, coloca el conjunto entre corchetes con el símbolo ^ antes del primer caracter:

```
Console.Write (Regex.Match ("quiz qwerty", "q[^aeiou]").Index);    // 5
```

Puedes especificar un rango de caracteres utilizando un guión. La siguiente expresión regular hace coincidir un movimiento de ajedrez:

```
Console.Write (Regex.Match ("b1-c4", @"[a-h]\d-[a-h]\d").Success);  // True
```

\d indica un caracter de dígito, por lo que \d coincidirá con cualquier dígito. \D coincide con cualquier caracter que no sea un dígito.

\w indica un caracter de palabra, que incluye letras, números y el guión bajo (_).

\W coincide con cualquier caracter que no sea de palabra. Esto también funciona correctamente para letras no inglesas, como el cirílico.

. coincide con cualquier caracter excepto \n (pero permite \r).

\p coincide con un caracter en una categoría específica, como {Lu} para letras mayúsculas o {P} para puntuación (listamos las categorías en la sección de referencia más adelante en el capítulo):

```
Console.Write (Regex.IsMatch ("Yes, please", @"\p{P}"));   // True
```

Encontraremos más usos para \d, \w y . cuando los combinemos con cuantificadores.

Cuantificadores

Los cuantificadores hacen coincidir un elemento un número específico de veces.

Quantifier	Meaning
*	Zero or more matches
+	One or more matches
?	Zero or one match
{*n*}	Exactly *n* matches
{*n*,}	At least *n* matches
{*n*,*m*}	Between *n* and *m* matches

El cuantificador * hace coincidir el caracter o grupo precedente cero o más veces. Lo siguiente coincide con cv.docx, junto con cualquier versión numerada del mismo archivo (por ejemplo, cv2.docx, cv15.docx):

```
Console.Write (Regex.Match ("cv15.docx", @"cv\d*\.docx").Success);  // True
```

Observa que debemos escapar el punto en la extensión del archivo utilizando una barra invertida.

Lo siguiente permite cualquier cosa entre cv y .docx y es equivalente a dir cv*.docx:

```
Console.Write (Regex.Match ("cvjoint.docx", @"cv.*\.docx").Success); // True
```

El cuantificador + hace coincidir el caracter o grupo precedente una o más veces. Por ejemplo:

```
Console.Write (Regex.Matches ("slow! yeah slooow!", "slo+w").Count);  // 2
```

El cuantificador {} hace coincidir un número específico (o rango) de repeticiones. Lo siguiente coincide con una lectura de presión arterial:

```
Regex bp = new Regex (@"\d{2,3}/\d{2,3}");
Console.WriteLine (bp.Match ("It used to be 160/110"));  // 160/110
Console.WriteLine (bp.Match ("Now it's only 115/75"));   // 115/75
```

Greedy Versus Lazy Quantifiers

Por defecto, los cuantificadores son ávidos (greedy), en contraposición a ser perezosos (lazy). Un cuantificador ávido se repite tantas veces como pueda antes de avanzar. Un cuantificador perezoso se repite tan pocas veces como pueda antes de avanzar. Puedes hacer que cualquier cuantificador sea perezoso añadiéndole el símbolo "?" al final. Para ilustrar la diferencia, consideremos el siguiente fragmento de HTML:

```
string html = "<i>By default</i> quantifiers are <i>greedy</i> creatures";
```

Supongamos que queremos extraer las dos frases en cursiva. Si ejecutamos el siguiente código:

```
foreach (Match m in Regex.Matches (html, @"<i>.*</i>"))
  Console.WriteLine (m);
```

el resultado no son dos coincidencias, sino una única coincidencia:

```
<i>By default</i> quantifiers are <i>greedy</i>
```

El problema es que nuestro cuantificador "*" se repite ávidamente tantas veces como pueda antes de hacer coincidir "</i>". Así que pasa directamente por la primera "</i>", deteniéndose solo en el "</i>" final (el último punto en el que el resto de la expresión aún puede hacer coincidir).

Si hacemos que el cuantificador sea perezoso, el "*" se detendrá en el primer punto en el que el resto de la expresión pueda hacer coincidir:

```
foreach (Match m in Regex.Matches (html, @"<i>.*?</i>"))
  Console.WriteLine (m);
```

Aquí está el resultado:

```
<i>By default</i>
<i>greedy</i>
```

Afirmaciones de ancho cero (Zero-Width Assertions)

El lenguaje de las expresiones regulares te permite establecer condiciones sobre lo que debe ocurrir antes o después de una coincidencia, a través de búsquedas hacia atrás (lookbehind), búsquedas hacia adelante (lookahead), anclas y límites de palabras. Estas se llaman afirmaciones de ancho cero porque no aumentan el ancho (o longitud) de la coincidencia en sí.

Búsqueda hacia adelante (lookahead) y búsqueda hacia atrás (lookbehind)

La construcción "(?=expr)" verifica si el texto que sigue coincide con expr, sin incluir expr en el resultado. Esto se llama búsqueda hacia adelante positiva. En el siguiente ejemplo, buscamos un número seguido de la palabra "millas":

```
Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles)"));

OUTPUT: 25
```

Observa que la palabra "millas" no se devuelve en el resultado, aunque se requiere para satisfacer la coincidencia.

Después de una vista previa exitosa, la coincidencia continúa como si la vista previa nunca hubiera ocurrido. Por lo tanto, si agregamos .* a nuestra expresión de esta manera:

```
Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles).*"));
```

el resultado sería "25 miles more".

La vista previa puede ser útil para imponer reglas para una contraseña segura. Supongamos que una contraseña debe tener al menos seis caracteres y contener al menos un dígito. Con una vista previa, podríamos lograr esto de la siguiente manera:

```
string password = "...";
bool ok = Regex.IsMatch (password, @"(?=.*\d).{6,}");
```

Esto primero realiza una vista previa para asegurarse de que haya un dígito en alguna parte de la cadena. Si se cumple, vuelve a su posición antes de que comenzara la vista previa y coincide con seis o más caracteres. (En "Cookbook Regular Expressions" en la página 999, incluimos un ejemplo más completo de validación de contraseñas).

Por otro lado, el contrario es la construcción de vista previa negativa, (?!expr). Esto requiere que la coincidencia no esté seguida por expr. La siguiente expresión coincide con "good", a menos que "however" o "but" aparezcan más adelante en la cadena:

```
string regex = "(?i)good(?!.*(however|but))";
Console.WriteLine (Regex.IsMatch ("Good work! But...",  regex));  // False
Console.WriteLine (Regex.IsMatch ("Good work! Thanks!", regex));  // True
```

La construcción (?<=expr) denota la vista previa positiva y requiere que haya una coincidencia precedida por una expresión especificada. La construcción opuesta, (?<!expr), denota la vista previa negativa y requiere que no haya una coincidencia precedida por una expresión especificada. Por ejemplo, lo siguiente coincide con "good", a menos que "however" aparezca antes en la cadena:

```
string regex = "(?i)(?<!however.*)good";
Console.WriteLine (Regex.IsMatch ("However good, we...", regex)); // False
Console.WriteLine (Regex.IsMatch ("Very good, thanks!", regex));  // True
```

Podríamos mejorar estos ejemplos agregando afirmaciones de límite de palabra, que presentaremos en breve.

Anclas (Anchors)

Los anclas ^ y $ coinciden con una posición específica. Por defecto:

^

Coincide con el inicio de la cadena

$

Coincide con el final de la cadena

 ^ tiene dos significados dependientes del contexto: un ancla y un negador de clase de caracteres.

$ tiene dos significados dependientes del contexto: un ancla y un indicador de grupo de reemplazo.

Por ejemplo:

```
Console.WriteLine (Regex.Match ("Not now", "^[Nn]o"));   // No
Console.WriteLine (Regex.Match ("f = 0.2F", "[Ff]$"));   // F
```

Cuando especificas RegexOptions.Multiline o incluyes (?m) en la expresión:

- ^ coincide con el inicio de la cadena o línea (justo después de un \n).
- $ coincide con el final de la cadena o línea (justo antes de un \n).

Hay un detalle al usar $ en modo multilínea: una nueva línea en Windows casi siempre se denota con \r\n en lugar de solo \n. Esto significa que para que $ sea útil en archivos de Windows, generalmente debes hacer coincidir el \r también, usando una vista previa positiva:

```
(?=\r?$)
```

La vista previa positiva asegura que \r no forme parte del resultado. Lo siguiente coincide con líneas que terminan en ".txt":

```
string fileNames = "a.txt" + "\r\n" + "b.docx" + "\r\n" + "c.txt";
string r = @".+\.txt(?=\r?$)";
foreach (Match m in Regex.Matches (fileNames, r, RegexOptions.Multiline))
  Console.Write (m + " ");

OUTPUT: a.txt c.txt
```

Lo siguiente coincide con todas las líneas vacías en la cadena s:

```
MatchCollection emptyLines = Regex.Matches (s, "^(?=\r?$)",
                                            RegexOptions.Multiline);
```

Lo siguiente coincide con todas las líneas que están vacías o contienen solo espacios en blanco:

```
MatchCollection blankLines = Regex.Matches (s, "^[ \t]*(?=\r?$)",
                                            RegexOptions.Multiline);
```

 Dado que una ancla coincide con una posición en lugar de un carácter, especificar solo una ancla coincide con una cadena vacía:

```
Console.WriteLine (Regex.Match ("x", "$").Length);   // 0
```

Límites de palabras (Word Boundaries)

La afirmación de límite de palabra \b coincide donde los caracteres de palabra (\w) se unen a:

- Caracteres que no son palabras (\W)
- El inicio/final de la cadena (^ y $)

\b se usa con frecuencia para coincidir con palabras completas:

```
foreach (Match m in Regex.Matches ("Wedding in Sarajevo", @"\b\w+\b"))
  Console.WriteLine (m);

Wedding
in
Sarajevo
```

Las siguientes afirmaciones resaltan el efecto de un límite de palabra:

```
int one = Regex.Matches ("Wedding in Sarajevo", @"\bin\b").Count; // 1
int two = Regex.Matches ("Wedding in Sarajevo", @"in").Count;     // 2
```

La siguiente consulta utiliza una vista previa positiva para devolver palabras seguidas de "(sic)":

```
string text = "Don't loose (sic) your cool";
Console.Write (Regex.Match (text, @"\b\w+\b\s(?=\(sic\))"));  // loose
```

Grupos

A veces, es útil separar una expresión regular en una serie de subexpresiones o grupos. Por ejemplo, considera la siguiente expresión regular que representa un número de teléfono de EE. UU., como 206-465-1918:

```
\d{3}-\d{3}-\d{4}
```

Supongamos que queremos separar esto en dos grupos: el código de área y el número local. Podemos lograr esto utilizando paréntesis para capturar cada grupo:

```
(\d{3})-(\d{3}-\d{4})
```

Luego recuperamos los grupos programáticamente:

```
Match m = Regex.Match ("206-465-1918", @"(\d{3})-(\d{3}-\d{4})");

Console.WriteLine (m.Groups[1]);   // 206
Console.WriteLine (m.Groups[2]);   // 465-1918
```

El grupo cero representa la coincidencia completa. En otras palabras, tiene el mismo valor que el Value de la coincidencia:

```
Console.WriteLine (m.Groups[0]);   // 206-465-1918
Console.WriteLine (m);             // 206-465-1918
```

Los grupos forman parte del lenguaje de expresiones regulares en sí mismo. Esto significa que puedes hacer referencia a un grupo dentro de una expresión regular. La sintaxis \n te permite indexar el grupo por su número de grupo n dentro de la expresión. Por ejemplo, la expresión (\w)ee\1 coincide con "deed" y "peep". En el siguiente ejemplo, encontramos todas las palabras en una cadena que comienzan y terminan con la misma letra:

```
foreach (Match m in Regex.Matches ("pop pope peep", @"\b(\w)\w+\1\b"))
  Console.Write (m + " ");  // pop peep
```

Los corchetes alrededor de `\w` instruyen al motor de expresiones regulares a almacenar la subcoincidencia en un grupo (en este caso, una sola letra) para que pueda ser utilizada más tarde. Nos referimos a ese grupo más tarde usando `\1`, lo que significa el primer grupo en la expresión.

Grupos con nombres

En una expresión larga o compleja, puede ser más fácil trabajar con grupos por nombre en lugar de índices. Aquí tienes una reescritura del ejemplo anterior, utilizando un grupo al que le asignamos el nombre 'letter':

```
string regEx =
  @"\b"             +  // word boundary
  @"(?'letter'\w)"  +  // match first letter, and name it 'letter'
  @"\w+"            +  // match middle letters
  @"\k'letter'"     +  // match last letter, denoted by 'letter'
  @"\b";               // word boundary

foreach (Match m in Regex.Matches ("bob pope peep", regEx))
  Console.Write (m + " ");  // bob peep
```

Así es como se le asigna un nombre a un grupo capturado:

```
(?'group-name'group-expr)  or  (?<group-name>group-expr)
```

Y así es como se hace referencia a un grupo:

```
\k'group-name'  or  \k<group-name>
```

El siguiente ejemplo coincide con un elemento XML/HTML simple (no anidado) buscando nodos de inicio y fin con un nombre coincidente:

```
string regFind =
  @"<(?'tag'\w+?).*>" +  // lazy-match first tag, and name it 'tag'
  @"(?'text'.*?)"     +  // lazy-match text content, name it 'text'
  @"</\k'tag'>";         // match last tag, denoted by 'tag'

Match m = Regex.Match ("<h1>hello</h1>", regFind);
Console.WriteLine (m.Groups ["tag"]);          // h1
Console.WriteLine (m.Groups ["text"]);         // hello
```

Permitir todas las posibles variaciones en la estructura XML, como elementos anidados, es más complejo. El motor de expresiones regulares de .NET tiene una extensión sofisticada llamada "matched balanced constructs" que puede ayudar con las etiquetas anidadas. Puedes encontrar más información sobre esto en internet y en el libro "Mastering Regular Expressions" (O'Reilly) de Jeffrey E. F. Friedl.

Reemplazar y dividir texto

El método `RegEx.Replace` funciona como `string.Replace`, pero utiliza una expresión regular.

El siguiente ejemplo reemplaza "cat" por "dog". A diferencia de `string.Replace`, "catapult" no se cambiará a "dogapult", porque hacemos coincidir en los límites de palabra:

```
string find = @"\bcat\b";
string replace = "dog";
Console.WriteLine (Regex.Replace ("catapult the cat", find, replace));

OUTPUT: catapult the dog
```

La cadena de reemplazo puede hacer referencia a la coincidencia original con la construcción de sustitución `$0`. El siguiente ejemplo envuelve los números dentro de una cadena entre corchetes angulares:

```
string text = "10 plus 20 makes 30";
Console.WriteLine (Regex.Replace (text, @"\d+", @"<$0>"));

OUTPUT: <10> plus <20> makes <30>
```

Puedes acceder a cualquier grupo capturado con `$1`, `$2`, `$3`, y así sucesivamente, o `${nombre}` para un grupo con nombre. Para ilustrar cómo esto puede ser útil, considera la expresión regular en la sección anterior que coincide con un elemento XML simple. Al reorganizar los grupos, podemos formar una expresión de reemplazo que mueve el contenido del elemento a un atributo XML:

```
string regFind =
  @"<(?'tag'\w+?).*>" +  // lazy-match first tag, and name it 'tag'
  @"(?'text'.*?)"     +  // lazy-match text content, name it 'text'
  @"</\k'tag'>";         // match last tag, denoted by 'tag'

string regReplace =
  @"<${tag}"          +  // <tag
  @"value="""         +  // value="
  @"${text}"          +  // text
  @"""/>";               // "/>

Console.Write (Regex.Replace ("<msg>hello</msg>", regFind, regReplace));
```

Aquí está el resultado:

```
 <msg value="hello"/>
```

Delegado MatchEvaluator

El método `Replace` tiene una sobrecarga que toma un delegado `MatchEvaluator`, que se invoca por coincidencia. Esto te permite delegar el contenido de la cadena de reemplazo al código de C# cuando el lenguaje de expresiones regulares no es lo suficientemente expresivo:

```
Console.WriteLine (Regex.Replace ("5 is less than 10", @"\d+",
                   m => (int.Parse (m.Value) * 10).ToString()) );

OUTPUT: 50 is less than 100
```

En "Cookbook Regular Expressions" en la página 999, mostramos cómo usar un `MatchEvaluator` para escapar correctamente los caracteres Unicode en HTML.

Splitting Text (División de texto)

La función estática Regex.Split es una versión más potente del método string.Split, utilizando una expresión regular que define el patrón separador. En este ejemplo, separamos una cadena en la que cualquier dígito cuenta como separador:

```
foreach (string s in Regex.Split ("a5b7c", @"\d"))
  Console.Write (s + " ");     // a b c
```

En este caso, el resultado no incluye los separadores en sí. Sin embargo, es posible incluir los separadores envolviendo la expresión en una búsqueda positiva hacia adelante. El siguiente ejemplo divide una cadena en palabras separadas en caso de que haya una transición de mayúsculas a minúsculas:

```
foreach (string s in Regex.Split ("oneTwoThree", @"(?=[A-Z])"))
  Console.Write (s + " ");    // one Two Three
```

Recetas de Expresiones Regulares (Cookbook Regular Expressions)

Recetas

Coincidir con números de seguro social/telefónicos de EE. UU.

```
string ssNum = @"\d{3}-\d{2}-\d{4}";

Console.WriteLine (Regex.IsMatch ("123-45-6789", ssNum));      // True

string phone = @"(?x)
  ( \d{3}[-\s] | \(\d{3}\)\s? )
    \d{3}[-\s]?
    \d{4}";

Console.WriteLine (Regex.IsMatch ("123-456-7890",   phone));   // True
Console.WriteLine (Regex.IsMatch ("(123) 456-7890", phone));   // True
```

Extraer pares de "nombre = valor" (uno por línea)

Ten en cuenta que esto comienza con la directiva multilinea (?m):

```
string r = @"(?m)^\s*(?'name'\w+)\s*=\s*(?'value'.*)\s*(?=\r?$)";

string text =
  @"id = 3
    secure = true
    timeout = 30";

foreach (Match m in Regex.Matches (text, r))
  Console.WriteLine (m.Groups["name"] + " is " + m.Groups["value"]);
id is 3 secure is true timeout is 30
```

Validación de contraseñas seguras

Lo siguiente verifica si una contraseña tiene al menos seis caracteres y si contiene un dígito, un símbolo o un signo de puntuación:

```
string r = @"(?x)^(?=.* ( \d | \p{P} | \p{S} )).{6,}";

Console.WriteLine (Regex.IsMatch ("abc12", r));     // False
Console.WriteLine (Regex.IsMatch ("abcdef", r));    // False
Console.WriteLine (Regex.IsMatch ("ab88yz", r));    // True
```

Líneas de al menos 80 caracteres

```
string r = @"(?m)^.{80,}(?=\r?$)";

string fifty = new string ('x', 50);
string eighty = new string ('x', 80);

string text = eighty + "\r\n" + fifty + "\r\n" + eighty;

Console.WriteLine (Regex.Matches (text, r).Count);   // 2
```

Analizando fechas/horas (N/N/N H:M:S AM/PM)

Esta expresión maneja una variedad de formatos de fechas numéricas y funciona tanto si el año va primero como si va al final. La directiva (?x) mejora la legibilidad al permitir espacios en blanco; la directiva (?i) desactiva la distinción entre mayúsculas y minúsculas (para el indicador opcional AM/PM). Luego, puedes acceder a cada componente de la coincidencia a través de la colección Groups:

```
string r = @"(?x)(?i)
 (\d{1,4}) [./-]
 (\d{1,2}) [./-]
 (\d{1,4}) [\sT]
 (\d+):(\d+):(\d+) \s? (A\.?M\.?|P\.?M\.?)?";

string text = "01/02/2008 5:20:50 PM";

foreach (Group g in Regex.Match (text, r).Groups)
  Console.WriteLine (g.Value + " ");
01/02/2008 5:20:50 PM 01 02 2008 5 20 50 PM
```

(Por supuesto, esto no verifica si la fecha/hora es correcta).

Coincidencia de números romanos

```
string r =
  @"(?i)\bm*"          +
  @"(d?c{0,3}|c[dm])" +
  @"(l?x{0,3}|x[lc])" +
  @"(v?i{0,3}|i[vx])" +
  @"\b";

Console.WriteLine (Regex.IsMatch ("MCMLXXXIV", r));   // True
```

Eliminación de palabras repetidas

Aquí capturamos un grupo con nombre llamado "dupe":

```
string r = @"(?'dupe'\w+)\W\k'dupe'";

string text = "In the the beginning...";
Console.WriteLine (Regex.Replace (text, r, "${dupe}"));

In the beginning
```

Conteo de palabras

```
string r = @"\b(\w|[-'])+\b";

string text = "It's all mumbo-jumbo to me";
Console.WriteLine (Regex.Matches (text, r).Count);   // 5
```

Coincidencia de un GUID

```
string r =
  @"(?i)\b"           +
  @"[0-9a-fA-F]{8}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{12}"  +
  @"\b";

string text = "Its key is {3F2504E0-4F89-11D3-9A0C-0305E82C3301}.";
Console.WriteLine (Regex.Match (text, r).Index);                    // 12
```

Análisis de una etiqueta XML/HTML (Parsing an XML/HTML tag)

Regex es útil para analizar fragmentos de HTML, especialmente cuando el documento puede estar imperfectamente formado:

```
string r =
  @"<(?'tag'\w+?).*>"  +  // lazy-match first tag, and name it 'tag'
  @"(?'text'.*?)"      +  // lazy-match text content, name it 'textd'
  @"</\k'tag'>";          // match last tag, denoted by 'tag'

string text = "<h1>hello</h1>";

Match m = Regex.Match (text, r);

Console.WriteLine (m.Groups ["tag"]);       // h1
Console.WriteLine (m.Groups ["text"]);      // hello
```

División de una palabra en notación camel case

Esto requiere un vistazo adelante positivo para incluir los separadores en mayúsculas:

```
string r = @"(?=[A-Z])";

foreach (string s in Regex.Split ("oneTwoThree", r))
  Console.Write (s + " ");    // one Two Three
```

Obteniendo un nombre de archivo válido

```
string input = "My \"good\" <recipes>.txt";

char[] invalidChars = System.IO.Path.GetInvalidFileNameChars();
string invalidString = Regex.Escape (new string (invalidChars));

string valid = Regex.Replace (input, "[" + invalidString + "]", "");
Console.WriteLine (valid);

My good recipes.txt
```

Escapando caracteres Unicode para HTML

```
string htmlFragment = "© 2007";

string result = Regex.Replace (htmlFragment, @"[\u0080-\uFFFF]",
                m => @"&#" + ((int)m.Value[0]).ToString() + ";");

Console.WriteLine (result);        // &#169; 2007
```

Desescapando caracteres en una cadena de consulta HTTP

```
string sample = "C%23 rocks";

string result = Regex.Replace (
    sample,
    @"%[0-9a-f][0-9a-f]",
    m => ((char) Convert.ToByte (m.Value.Substring (1), 16)).ToString(),
    RegexOptions.IgnoreCase
);

Console.WriteLine (result);   // C# rocks
```

Analizando términos de búsqueda de Google desde un registro de estadísticas web

Deberías usar esto en conjunto con el ejemplo anterior para desescapar caracteres en la cadena de consulta:

```
string sample =
  "http://google.com/search?hl=en&q=greedy+quantifiers+regex&btnG=Search";

Match m = Regex.Match (sample, @"(?<=google\..+search\?.*q=).+?(?=(&|$))");

string[] keywords = m.Value.Split (
  new[] { '+' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string keyword in keywords)
  Console.Write (keyword + " ");       // greedy quantifiers regex
```

Regular Expressions Language Reference

Tables 25-2 through 25-12 summarize the regular expressions grammar and syntax supported in the .NET implementation.

Table 25-2. Character escapes

Escape code sequence	Meaning	Hexadecimal equivalent
\a	Bell	\u0007
\b	Backspace	\u0008
\t	Tab	\u0009
\r	Carriage return	\u000A
\v	Vertical tab	\u000B
\f	Form feed	\u000C
\n	Newline	\u000D
\e	Escape	\u001B
nnn	ASCII character *nnn* as octal (e.g., \n052)	
\x*nn*	ASCII character *nn* as hex (e.g., \x3F)	
\c*l*	ASCII control character *l* (e.g., \cG for Ctrl-G)	
\u*nnnn*	Unicode character *nnnn* as hex (e.g., \u07DE)	
symbol	A nonescaped symbol	

Special case: within a regular expression, \b means word boundary, except in a [ ] set, in which \b means the backspace character.

Table 25-3. Character sets

Expression	Meaning	Inverse ("not")
`[abcdef]`	Matches a single character in the list	`[^abcdef]`
`[a-f]`	Matches a single character in a *range*	`[^a-f]`
`\d`	Matches a decimal digit Same as `[0-9]`	`\D`
`\w`	Matches a *word* character (by default, varies according to `CultureInfo.CurrentCulture`; for example, in English, same as `[a-zA-Z_0-9]`)	`\W`
`\s`	Matches a whitespace character Same as `[\n\r\t\f\v ]`	`\S`
`\p{category}`	Matches a character in a specified *category* (see Table 25-4)	`\P`
`.`	(Default mode) Matches any character except `\n`	`\n`
`.`	(`SingleLine` mode) Matches any character	`\n`

Table 25-4. Character categories

Quantifier	Meaning
`\p{L}`	Letters
`\p{Lu}`	Uppercase letters
`\p{Ll}`	Lowercase letters
`\p{N}`	Numbers
`\p{P}`	Punctuation
`\p{M}`	Diacritic marks
`\p{S}`	Symbols
`\p{Z}`	Separators
`\p{C}`	Control characters

Table 25-5. Quantifiers

Quantifier	Meaning
`*`	Zero or more matches
`+`	One or more matches
`?`	Zero or one match
`{n}`	Exactly *n* matches
`{n,}`	At least *n* matches
`{n,m}`	Between *n* and *m* matches

El sufijo "?" se puede aplicar a cualquiera de los cuantificadores para hacerlos "perezosos" en lugar de "codiciosos".

Table 25-6. Substitutions

Expression	Meaning
`$0`	Substitutes the matched text
`$group-number`	Substitutes an indexed *`group-number`* within the matched text
`${group-name}`	Substitutes a text *`group-name`* within the matched text

Las sustituciones se especifican solo dentro de un patrón de reemplazo.

Table 25-7. Zero-width assertions

Expression	Meaning
`^`	Start of string (or line in *multiline* mode)
`$`	End of string (or line in *multiline* mode)
`\A`	Start of string (ignores *multiline* mode)

Expression	Meaning
`\z`	End of string (ignores *multiline* mode)
`\Z`	End of line or string
`\G`	Where search started
`\b`	On a word boundary
`\B`	Not on a word boundary
`(?=expr)`	Continue matching only if expression *`expr`* matches on right (*positive lookahead*)
`(?!expr)`	Continue matching only if expression *`expr`* doesn't match on right (*negative lookahead*)
`(?<=expr)`	Continue matching only if expression *`expr`* matches on left (*positive lookbehind*)
`(?<!expr)`	Continue matching only if expression *`expr`* doesn't match on left (*negative lookbehind*)
`(?>expr)`	Subexpression *`expr`* is matched once and not backtracked

Table 25-8. Grouping constructs

Syntax	Meaning
`(expr)`	Capture matched expression *`expr`* into indexed group
`(?number)`	Capture matched substring into a specified group *`number`*
`(?'name')`	Capture matched substring into group *`name`*
`(?'name1-name2')`	Undefine *`name2`* and store interval and current group into *`name1`*; if *`name2`* is undefined, matching backtracks
`(?:expr)`	Noncapturing group

Table 25-9. Back references

Parameter syntax	Meaning
`\index`	Reference a previously captured group by *`index`*
`\k<name>`	Reference a previously captured group by *`name`*

Table 25-10. Alternation

Expression syntax	Meaning
|	Logical *or*
(?(*expr*)*yes*|*no*)	Matches *yes* if expression matches; otherwise, matches *no* (*no* is optional)
(?(*name*)*yes*|*no*)	Matches *yes* if named group has a match; otherwise, matches *no* (*no* is optional)

Table 25-11. Miscellaneous constructs

Expression syntax	Meaning
(?#*comment*)	Inline comment
#*comment*	Comment to end of line (works only in `IgnorePatternWhitespace` mode)