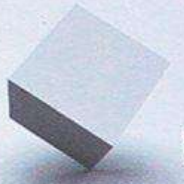
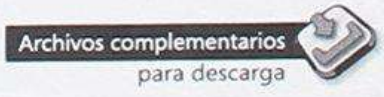


Recursosinformáticos

C# 7
y Visual Studio 2017
Los fundamentos del lenguaje

Sébastien PUTIER



Contenido

C# 7 y Visual Studio 2017

Titulo, autor… Los fundamentos del lenguaje

Este libro se dirige a aquellos **desarrolladores** que deseen dominar el desarrollo de aplicaciones .Net con **el lenguaje C# en su versión 7**.

Tras recorrer el panorama de la **plataforma .Net** y describir las herramientas proporcionadas por el entorno **Visual Studio 2017**, el lector descubrirá progresivamente los **elementos clave del lenguaje C#** y la **programación orientada a objetos**, hasta poder aplicar estos conceptos al **desarrollo de aplicaciones Windows con WPF**. Una iniciación a las técnicas de **depuración**con Visual Studio le permitirá perfeccionar su dominio de la herramienta.

El desarrollo de aplicaciones cliente-servidor se aborda, a continuación, mediante el tema del **acceso a datos con ADO.Net**. Se realiza una descripción completa de **Linq** a través de ejemplos concretos que muestran cómo sus funcionalidades simplifican la manipulación de los datos. Su uso está sobretodo presente en el ámbito del acceso al contenido de una base de datos SQL Server con **Entity Framework**. A continuación, se dedica un capítulo a la manipulación de datos en **formato XML**, que permite intercambiar datos entre aplicaciones de una manera sencilla y estandarizada. El final del ciclo de desarrollo se aborda mediante el desarrollo de una aplicación con las tecnologías **Windows Installer** y **ClickOnce**.

Para ayudar al lector en su aprendizaje se proporciona un **glosario** que resume la utilidad de las palabras clave de C# que se abordan en el libro.

Los ejemplos que se exponen en estas páginas están disponibles para su descarga en esta página.

Los capítulos del libro:
Prólogo – La plataforma .NET – Visual Studio 2017 – La organización de una aplicación – Las bases del lenguaje – Programación orientada a objetos con C# – Depuración y gestión de errores – Desarrollo de aplicaciones Windows – El acceso a datos – LINQ – XML – Despliegue – Glosario

---

Sébastien PUTIER

Consultor y formador desde hace varios años, **Sébastien PUTIER** aporta su conocimiento en la implementación de soluciones Windows, Web y móviles mediante la plataforma .Net desde sus primeras versiones. Está certificado técnicamente (MCPD – MCSD) y pedagógicamente (MCT) por Microsoft. A lo largo de este libro, transmite toda su experiencia acerca del lenguaje C# para que el lector adquiera los conceptos fundamentales y sea capaz de sacar el máximo partido posible de las funcionalidades ofrecidas por la plataforma .Net.

Prologo-Introducción

Desde la aparición de C# con la primera versión de la plataforma .NET, el lenguaje ha conocido una importante evolución hasta alcanzar su séptima versión oficial. Las múltiples funcionalidades que permite utilizar, así como su sintaxis, hacen de él el lenguaje de referencia en el universo .NET.

El lugar de C# en el mundo del desarrollo ha progresado con el tiempo, hasta alcanzar la situación actual: C# es un lenguaje "a tener en cuenta". Permite desarrollar numerosas plataformas de hardware y software (particularmente gracias al proyecto Mono) y se utiliza en muchas empresas que ven en él una herramienta de producción ideal.

Este libro está dirigido a todos aquellos desarrolladores que deseen aprender a desarrollar aplicaciones con la ayuda de C# y Visual Studio 2017.

En primer lugar, **descubrirá la plataforma .NET** a través de su historia y la descripción de sus componentes, a continuación verá cómo instalarla, así como la presentación del entorno de desarrollo **Visual Studio 2017**. La **estructuración de soluciones y proyectos** se detalla también para que pueda familiarizarse con las herramientas que existen a su disposición.

Una vez superada esta etapa, descubrirá los **elementos fundamentales de C#** y su aplicación a la **programación orientada a objetos** con el objetivo de utilizar mejor las funcionalidades ofrecidas por la plataforma .NET. Para poder diagnosticar y resolver los distintos errores que pueda cometer en sus desarrollos, se dedica un capítulo a la **problemática de depuración y de gestión de errores**.

Una vez adquiridas estas nociones seremos capaces de dominar el diseño de **aplicaciones de ventanas utilizando la tecnología WPF**. Durante sus futuros desarrollos probablemente necesite, además de una interfaz gráfica de ensueño, trabajar con datos almacenados en bases de datos. Los dos siguientes capítulos le permitirán dominar este aspecto abordando **ADO.NET** y **LINQ**. ADO.NET proporciona las herramientas fundamentales para la manipulación de datos, en particular mediante consultas SQL. LINQ, por su parte, es una API de más alto nivel que ofrece la posibilidad de realizar procesamientos complejos sobre juegos de datos almacenados en memoria o en una base de datos.

El formato **XML** es hoy en día un estándar que permite intercambiar datos entre aplicaciones o a través de la Web. Por ello, le dedica un capítulo para que sea capaz de manipular la estructura y el contenido de documentos XML.

Por último, se presenta un capítulo que aborda una parte importante del ciclo de vida de una aplicación: el **despliegue**. Se describen dos tecnologías integradas en Visual Studio para que pueda disponer de las claves que le permitirán trabajar lo mejor posible: **InstallShield** y **ClickOnce**.

Para ayudarle en su aprendizaje y su perfeccionamiento, el último capítulo del libro ofrece un **glosario** que agrupa las distintas palabras clave del lenguaje. Cada una de ellas se acompaña de una rápida descripción y de un ejemplo de uso que le permitirán, de esta manera, tener siempre a mano una **referencia rápida y accesible**.

La plataforma .net - Introducción

Microsoft pone a su disposición, mediante la plataforma .NET, un conjunto de herramientas y tecnologías que permiten desarrollar aplicaciones destinadas a plataformas muy variadas:

- Aplicaciones Windows (aplicaciones de ventanas, aplicaciones de consola, servicios Windows)
- Aplicaciones y servicios web
- Aplicaciones para smartphones
- Aplicaciones orientadas a tabletas táctiles
- Aplicaciones para sistemas embebidos

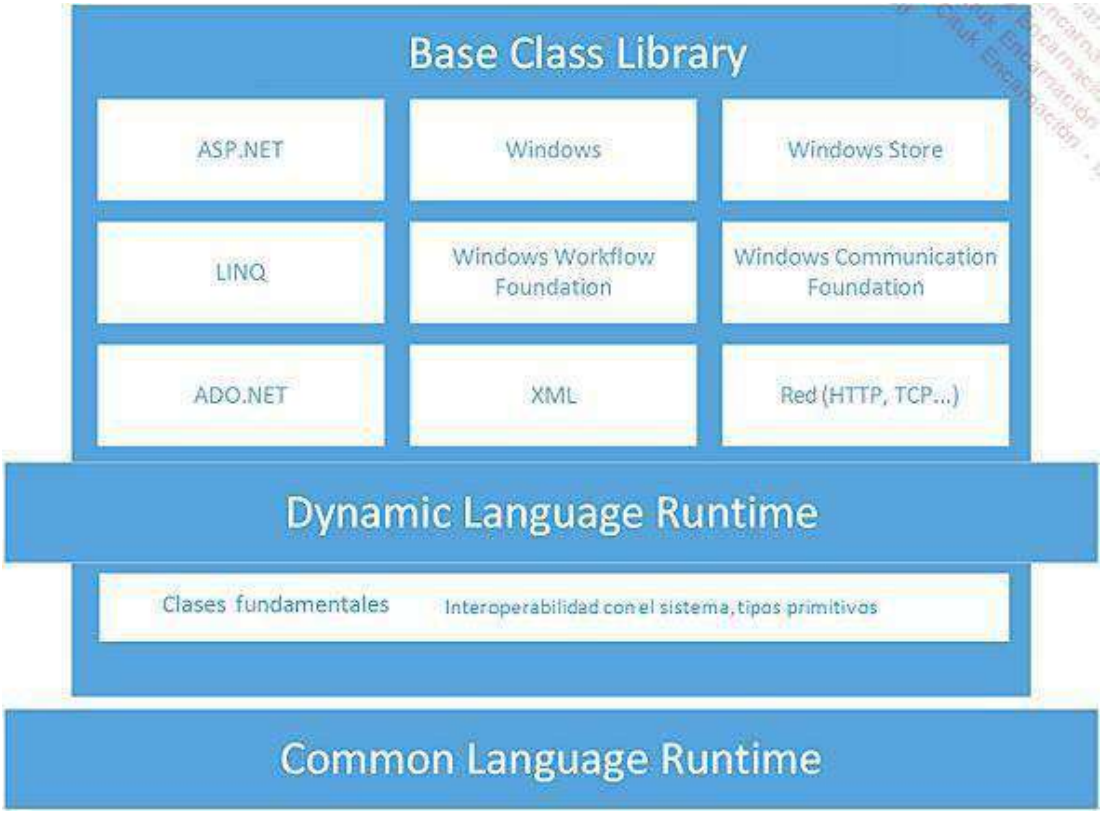
Estos distintos tipos de aplicaciones pueden desarrollarse gracias a un elemento común: el **framework .NET**. Este framework es una solución de software que incluye varios componentes dedicados al desarrollo y ejecución de las aplicaciones. Lo facilita Microsoft con el sistema operativo Windows y está disponible para otros sistemas mediante soluciones de software de terceros, como Mono.

Los elementos que forman el núcleo del framework .NET son el **Common Language Runtime (CLR)**, el **Dynamic Language Runtime (DLR)** y la **Base Class Library (biblioteca de clases básicas)**.

El Common Language Runtime es un entorno que permite ejecutar código .NET y que asegura la gestión de la memoria. El código gestionado por el CLR se denomina generalmente código manejado.

El Dynamic Language Runtime es un complemento al Common Language Runtime, desarrollado en C#, que provee capacidades de ejecución de código dinámico. Gracias a él, lenguajes dinámicos como PHP o Python están soportados por el framework .NET.

La Base Class Library es un conjunto de clases que exponen funcionalidades habituales y que pueden utilizarse en cualquier tipo de aplicación. Está formada por varios bloques independientes que se han ido agregando progresivamente a lo largo de las ediciones del framework .NET.


Base Class Library
ASP.NET
Windows
Windows Store
LINQ
Windows Workflow Foundation
Windows Communication Foundation
ADO.NET
XML
Red (HTTP, TCP...)
Dynamic Language Runtime
Clases fundamentales
Interoperabilidad con el sistema, tipos primitivos
Common Language Runtime

Historia

Desde sus inicios, Windows expone funcionalidades a los desarrolladores a través de interfaces de programación (en inglés API, de *Application Programming Interface*) que les permiten crear aplicaciones. Con la complejidad que ha adquirido Windows a lo largo de los años, distintas API han ido apareciendo y evolucionando:

- **API Windows (Win16, Win32 y Win64)**: estas API forman el núcleo de Windows y las utiliza directamente el sistema. Están destinadas a utilizarse al más bajo nivel, con el lenguaje C. Win16 apareció con Windows 3.1, mientras que Win32 se utiliza desde Windows 95 y Win64 está integrado en los sistemas de 64 bits desde Windows XP.
- **MFC** (*Microsoft Foundation Classes*) existe desde 1992 y encapsula las API Win16 y Win32 en una estructura orientada a objetos en C++.
- **COM** (*Component Object Model*) aparece poco después de las MFC para responder a una problemática de comunicación entre procesos. A continuación, la librería **ATL** (*Active Template Library*) encapsuló las API COM para simplificar su uso.

La plataforma .NET apareció a principios de los años 2000 para dar respuesta al problema de la complejidad de estas API. Las unifica y las moderniza, permitiendo su uso con cualquier lenguaje que posea una implementación compatible. Visual C# se creó para la plataforma .NET con el objetivo de disponer de un lenguaje capaz de explotar todas las capacidades que ofrece.

La unificación de las API y de los modelos de desarrollo también ha tocado al mundo de la Web. En efecto, desde mediados de los años 90, Microsoft facilita del lado del servidor, el motor de script **ASP** (*Active Server Page*), que se dejó de usar en beneficio de un componente integrado en el framework .NET: **ASP.NET**. Este ha permitido mejorar el rendimiento de las aplicaciones web introduciendo código compilado y no interpretado. También ha mejorado su mantenibilidad remplazando la programación procedural típica de los lenguajes de script con un modelo de programación de eventos y orientado a objetos.

Desde su aparición, han aparecido trece versiones diferentes de la plataforma .NET, de las cuales siete aportan modificaciones o funcionalidades mayores. A continuación se resume su evolución (de manera no exhaustiva).

Versión 1.0

Fecha de aparición: febrero de 2002

IDE asociado: Visual Studio .NET 2002

Funcionalidades asociadas:

- Desarrollo para Windows con los WinForms
- Desarrollo web con ASP.NET

Versión 1.1

Fecha de aparición: abril de 2003

IDE asociado: Visual Studio .NET 2003

Funcionalidades asociadas:

- Numerosos cambios en la API.
- Se incluyen proveedores de datos Oracle y ODBC en el componente ADO.NET (antes disponible como un add-on).
- Se incluyen controles visuales para el desarrollo en ASP. NET (antes disponible como un add-on).

- Soporte del protocolo IPv6.
- Primera versión de .NET Compact Framework destinado a dispositivos móviles.

Versión 2.0

Fecha de aparición: enero de 2006

IDE asociado: Visual Studio 2005

Funcionalidades asociadas:

- Soporte de sistemas operativos de 64 bits.
- Soporte de clases parciales que permiten separar entre el código generado y el código escrito por el desarrollador.
- Soporte de métodos anónimos.
- Introducción de los tipos genéricos que permiten escribir código capaz de adaptarse a distintos tipos de datos.

Versión 3.0

Fecha de aparición: noviembre de 2006

IDE asociado: Visual Studio 2005

Funcionalidades asociadas:

Cuatro nuevos bloques principales:

- WPF (*Windows Presentation Foundation*) para la definición de interfaces gráficas.
- WCF (*Windows Communication Foundation*) para la comunicación para la interoperabilidad entre aplicaciones.
- WF (*Windows Workflow Foundation*), para definir flujos de trabajo automatizados.
- Windows CardSpace, para la gestión de la identidad por autenticación única (actualmente obsoleto).

Versión 3.5

Fecha de aparición: noviembre de 2007

IDE asociado: Visual Studio 2008

Funcionalidades asociadas:

- Inferencia de tipos
- LINQ para la consulta de datos

Versión 4.0

Fecha de aparición: abril de 2010

IDE asociado: Visual Studio 2010

Funcionalidades asociadas:

- Introducción del DLR (*Dynamic Language Runtime*) para la interfaz con lenguajes dinámicos.
- Se incluyen las Parallel Extensions para gestionar el multithreading y la ejecución paralela de código.
- Mejora del rendimiento del Garbage Collector.

Versión 4.5

Fecha de aparición: agosto de 2012

IDE asociado: Visual Studio 2012

Funcionalidades asociadas:

- Soporte de aplicaciones para Windows 8
- Extensión de C# y VB.NET para incluir el asincronismo directamente en los lenguajes
- Nueva API para la comunicación HTTP
- Mejora del rendimiento web con la minificación y el asincronismo

Versión 4.5.1

Fecha de aparición: octubre de 2013

IDE asociado: Visual Studio 2013

Funcionalidades asociadas:

- Soporte de aplicaciones para Windows 8.1

Versión 4.5.2

Fecha de aparición: mayo de 2014

IDE asociado: Visual Studio 2013

Funcionalidades asociadas:

- Nuevas API para ASP. NET

Versión 4.6

Fecha de aparición: verano de 2015

IDE asociado: Visual Studio 2015

Funcionalidades asociadas:

- Unificación de las API Web en el framework MVC 6
- .NET Core
- Compilación de las aplicaciones Windows Store en código nativo con .NET Native

- Nuevo compilador "Just-in-time" con mejor rendimiento: RyuJIT.
- Actualización de Windows Forms y WPF para el soporte a monitores con una mayor densidad de píxeles (High DPI).
- Soporte de nuevos algoritmos criptográficos.

Versión 4.6.1

Fecha de aparición: noviembre de 2015

IDE asociado: Visual Studio 2015

Funcionalidades asociadas:

- Mejoras en WPF (corrección ortográfica integrada, funcionalidades táctiles…).
- Mejora de rendimiento y de estabilidad.

Versión 4.6.2

Fecha de aparición: marzo de 2016

IDE asociado: Visual Studio 2015

Funcionalidades asociadas:

- Mejora de la criptografía.
- Mejora del soporte a pantallas con una gran densidad de píxeles.

Versión 4.7

Fecha de aparición: mayo de 2017

IDE asociado: Visual Studio 2017

Funcionalidades asociadas:

- Mejora de la criptografía.
- Mejora del protocolo de seguridad TLS.
- Nuevas API WPF dedicadas a la impresión.
- Mejora del soporte a monitores de gran densidad de píxeles.

El Common Language Runtime (CLR)

La ejecución de las aplicaciones Windows tradicionales la gestiona el propio sistema operativo. Estos programas se generan mediante un compilador que transforma instrucciones escritas en un lenguaje de programación como C o C++ en un archivo binario que contiene instrucciones específicas de un sistema operativo particular y una arquitectura de procesador específica. Con esta configuración es preciso generar varios ejecutables para soportar las distintas configuraciones de hardware y de software de los usuarios.

Una solución a esta problemática consiste en generar un archivo ejecutable cuyo código sea genérico, independiente del entorno de ejecución. En este contexto, su ejecución no puede confiarse directamente al sistema operativo dado que no es capaz de procesar el código genérico. Es necesario insertar un componente de software en la cadena de ejecución, entre la aplicación y el sistema, para traducir el código genérico en instrucciones adaptadas a la máquina. Un componente lógico de este tipo se denomina **máquina virtual**.

Este es exactamente el principio de funcionamiento que utiliza la plataforma .NET. La etapa de compilación de una aplicación escrita con Visual C# o Visual Basic .NET produce un archivo ejecutable cuyo contenido está, esencialmente, escrito en un lenguaje genérico llamado **Microsoft Intermediate Language** (o **MSIL**). Durante su ejecución, el **Common Language Runtime** (**CLR**) se encarga de procesar este código y realiza el rol de máquina virtual. Traduce al vuelo y bajo demanda las secciones de código que deben ejecutarse para optimizar los recursos utilizados para asegurar esta transformación.

El uso de esta técnica requiere la instalación de la máquina virtual en los distintos puestos de trabajo antes de ejecutar la aplicación. Microsoft proporciona una implementación de esta máquina virtual para todos los sistemas operativos Windows y, desde junio de 2016, para los sistemas Linux y Mac OS gracias a la iniciativa .NET Core. El proyecto Mono proporciona también, desde hace varios años, una implementación open source de este componente de software.

Además de esta operación de traducción, el Common Language Runtime proporciona una serie de servicios relativos a la ejecución del código generado. A continuación se muestra un esquema de las funcionalidades del CLR.

Class Loader

Gestiona la creación y la carga en memoria de las instancias de las clases.

MSIL To Native Compiler

Este servicio convierte el código intermedio en código nativo.

Code Manager

Gestiona la ejecución del código.

Garbage Collector

Proporciona una gestión automática de la memoria supervisando los objetos de la aplicación y liberando la memoria de las instancias de clases que ya no se utilizan y son inaccesibles.

Security Engine

Asegura la seguridad del sistema verificando los permisos de ejecución a nivel de código, de carpeta de aplicación y de máquina.

Debug Engine

Permite depurar la aplicación.

Type Checker

Verifica la compatibilidad de los distintos tipos utilizados en la aplicación para proveer una seguridad a nivel del tipado.

Exception Manager

Facilita la gestión individual de las excepciones que pueden elevarse por la aplicación.

Thread Support

Facilita una encapsulación manejada de los threads, permitiendo así el desarrollo de aplicaciones multithread que aprovechen todos los servicios suministrados por el CLR.

COM Marshaler

Permite establecer y realizar la comunicación entre una aplicación .NET y un componente COM, en particular realizando conversiones de tipos de datos automáticas si fuera necesario.

Base Class Library Support

Facilita acceso al conjunto de clases de la Class Library.

La Base Class Library (BCL)

El framework .NET incluye un conjunto de tipos de datos que permiten ofrecer rápidamente una solución a muchos de los problemas que podemos encontrar a lo largo del desarrollo de una aplicación.

Estos tipos de datos están organizados de manera jerárquica. Cada nivel de la jerarquía está definido por un **espacio de nombres** (**namespace**) que identifica a un grupo de tipos. Los espacios de nombres se nombran concatenando el nombre del padre, el símbolo "." y su propio nombre y todo ello hace referencia a un conjunto de funcionalidades que se proveen de manera conjunta. Por ejemplo, el espacio de nombres `System.Xml` agrupa un conjunto de clases que permiten procesar flujos de datos en formato XML.

La BCL contiene varios miles de tipos y es muy probable que nunca utilice gran parte de ellos en sus desarrollos. Algunos son, en efecto, muy específicos y se utilizan en muy pocos contextos. Así ocurre con todos los tipos definidos en los espacios de nombres `Microsoft.Build` o `System.CodeDom`, que permiten respectivamente interactuar con el motor MSBuild y generar código C# o Visual Basic .NET.

Entre los espacios de nombres más utilizados podemos destacar:

`System`

Se trata del espacio de nombres raíz para las funcionalidades suministradas por el sistema operativo.
Contiene, en particular, la definición de los tipos primitivos alfanuméricos y numéricos.

`System.Data`

Contiene el conjunto de tipos que permiten acceder a las bases de datos.

`System.IO`

Este espacio de nombres contiene los tipos que permiten acceder a flujos de datos o a archivos.

El Dynamic Language Runtime (DLR)

El Dynamic Language Runtime es una subcapa del Common Language Runtime, aparecida con la versión 4.0 de la plataforma .NET, que suministra servicios que facilitan la implementación y el uso de lenguajes dinámicos en un entorno .NET.

Desde la aparición del DLR es posible definir variables .NET de tipo dinámico, es decir, que no esté prefijado en el momento de la escritura del código. El uso de esta funcionalidad puede resultar extremadamente práctico cuando se trata de interactuar con aplicaciones escritas con lenguajes dinámicos o cuando se quieren utilizar ciertos componentes COM. Los tipos de datos devueltos por estos elementos de software no se conocen en el momento de la escritura del código, de modo que es difícil utilizarlos en lenguajes cuyo tipado sea puramente estático.

La existencia del Dynamic Language Runtime permite, en consecuencia, suministrar una **implementación de lenguajes dinámicos** como Python o Ruby para la plataforma .NET. Por otro lado, estos dos lenguajes ya están implementados en .NET con los nombres **IronPython** e **IronRuby**.

También es posible, gracias al DLR, crear aplicaciones .NET que permitan implementar scripts. Es decir, que sea posible escribir código compatible con .NET, con IronPython por ejemplo, y ejecutarlo en el interior de una aplicación Visual C# de la misma manera que es posible escribir scripts ejecutables en las aplicaciones Microsoft Access, 3DS Max o mIRC.

Los tipos .NET dedicados a la escritura del código dinámico están localizados en el espacio de nombres `System.Dynamic`.

Evolución de la plataforma

A lo largo de las distintas versiones, el entorno .NET se ha enriquecido con numerosas funcionalidades, entre ellas algunas eran/son realmente innovadoras. Las aplicaciones basadas en la plataforma pueden ejecutarse en equipos de escritorio, tabletas, navegadores web, smartphones o incluso en sistemas electrónicos embebidos. El número de desarrolladores que utilizan los lenguajes basados en el CLR también ha aumentado mucho hasta alcanzar varios millones.

En este punto de madurez se plantean nuevas problemáticas, tanto en el seno de los equipos de desarrollo de Microsoft como en la comunidad de desarrolladores que utilizan esta tecnología. Encontramos, en particular, entre estas preocupaciones la portabilidad del código o la necesidad de modernizar los compiladores C# y VB.NET.

Como respuesta a estos problemas, los equipos de desarrollo del framework .NET han definido e implementado el bloque .NET Core. También se han puesto manos a la obra y han escrito un compilador completamente nuevo para C# y VB.NET: Roslyn.

1. .NET Core

El framework .NET se ha diseñado como un bloque único, siempre desplegado en su conjunto. Su núcleo contiene el uso de API específicas para ciertos tipos de aplicaciones y, por consiguiente, los entornos que no soportan estas API deben utilizar un subconjunto de ese núcleo central. Las diferencias entre los tipos de aplicaciones desarrolladas con .NET han provocado problemas de factorización y de portabilidad del código.

La solución que aporta Microsoft a este problema se denomina .NET Core. Detrás de este nombre se esconde una rama del framework .NET cuyo objeto es ofrecer una mayor modularidad rompiendo la arquitectura monolítica de su biblioteca de clases. El código se ha refactorizado para obtener bloques más pequeños, cada uno centrado en una funcionalidad particular.

El modo de distribución también se ha revisado, pues cada porción de .NET Core se publica mediante un paquete NuGet. Esto permite una evolución más rápida de cada bloque. Cada equipo de desarrollo puede, también, utilizar una versión específica de un paquete y desplegarla con su aplicación.

Algunos paquetes están actualmente disponibles para todas las plataformas, como el bloque Entity Framework, pero el modo de funcionamiento basado por completo en .NET Core solo está disponible, de momento, para ASP.NET Core, las aplicaciones Windows 10 y las .NET Core Console Applications (aplicaciones de consola basadas específicamente en .NET Core).

Además de la portabilidad entre tipos de aplicaciones, Microsoft ha tenido en cuenta también la portabilidad entre sistemas operativos. El Core CLR es una versión compatible con Linux y Mac OS X del CLR integrado en el framework .NET. Las aplicaciones .NET están ahora soportadas oficialmente en estas plataformas.

2. .NET Compiler Platform: Roslyn

Roslyn es el nombre clave de la plataforma de compilación que provee Microsoft con el SDK del framework .NET desde la versión 4.6. Este sistema reemplaza al compilador C# de .NET 4.5, cuyo funcionamiento seguía el modelo tradicional de los compiladores: "se proporcionan archivos como entrada y, como por arte de magia, se construye una salida basada en uno o varios archivos".

Con Roslyn, Microsoft ha querido salir de este sistema de tipo "caja negra", transformando el compilador en un proveedor de datos. La API proporcionada con Roslyn permite recuperar y manipular el resultado del procesamiento realizado por cada proceso de la cadena de compilación:

- Análisis sintáctico
- Análisis semántico
- Escritura del código MSIL

La disponibilidad de estos datos abre la puerta a escenarios avanzados entre los cuales podemos describir la integración de C# en una aplicación como lenguaje de script compilado dinámicamente, o la extensión de un entorno de desarrollo para C#. También es posible imaginar herramientas empresariales que permitan validar la calidad del código escrito, generar la documentación técnica o compilar el código generado para integrarlo en una cadena de compilación más compleja.

3. .NET en el mundo open source

Desde hace algunos años Microsoft trabaja con el mundo open source, en particular en la integración de soluciones basadas en sistemas Linux en Azure. Más recientemente, se han desplegado las tecnologías Redis o Docker sobre esta plataforma, pero la integración de Microsoft en el paisaje open source no se limita al cloud Azure. La firma posee una división dedicada, Microsoft Open Tech, que trabaja en proyectos como Apache Cordova o la creación de nuevos estándares para la Web con el W3C.

El ecosistema .NET también está implicado: Microsoft ha anunciado la creación de la .NET Foundation tras la conferencia BUILD 2014. Esta estructura está dedicada al desarrollo de proyectos open source basados en la plataforma .NET. Encontramos, entre ellos, frameworks iniciados por desarrolladores independientes como MVVM Light Toolkit, o por empresas, como las librerías Xamarin.Auth y Xamarin.Mobile.

Ciertos proyectos centrales para Microsoft se han situado también bajo la gestión de esta fundación: .NET Core y Roslyn forman parte de ella, al igual que Entity Framework o ASP.NET Core. De este modo, su desarrollo está controlado por Microsoft, pero la comunidad puede, según el caso, proponer correcciones de bugs, de funcionalidades, o incluso intervenir directamente en los proyectos. Con esta iniciativa, Microsoft sitúa la plataforma .NET en la posición de elemento importante en el mundo libre, e implica a un gran número de desarrolladores en torno a sus tecnologías.

Una primera aplicación con Visual C#

Para descubrir Visual C#, estudiaremos a continuación la escritura de una aplicación muy sencilla así como el proceso de compilación del código. A esta primera fase le seguirá una etapa de análisis del ejecutable generado.

1. Creación

Una aplicación Visual C# se construye a partir de archivos de código fuente. Estos archivos se escriben, generalmente, con la ayuda de un entorno de desarrollo integrado como Visual Studio o, si está más familiarizado con herramientas open source, MonoDevelop o SharpDevelop.

Estos archivos de código fuente no son más que archivos de texto. Es perfectamente posible editarlos con la ayuda de la aplicación Bloc de Notas de Windows o con cualquier otro programa que permita abrir y modificar archivos de texto. La herramienta utilizada para editar estos archivos no debe, no obstante, agregar código de representación en la página como harían aplicaciones de procesamiento de textos como Word, WordPad o LibreOffice Writer.

El primer ejemplo que veremos en este libro es una aplicación que muestra el tradicional "¡Hola Mundo!" en una ventana de línea de comandos. Para escribir el código fuente correspondiente es necesario crear un archivo de texto con el editor que prefiera.

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("¡Hola Mundo!");
    }
}
```

Este archivo debe guardarse con un nombre cuyo sufijo sea la extensión `.cs`. Esta extensión no es obligatoria para la compilación del programa, pero sí se corresponde con las convenciones utilizadas en el desarrollo de aplicaciones con C#.

Detallemos a continuación el contenido de este código fuente:

`using System;`

Esta línea indica que el código utiliza uno o varios tipos localizados en el espacio de nombres System. Sin escribir esta línea, sería necesario utilizar el nombre plenamente cualificado de cada uno de estos tipos para poder utilizarlo. En nuestro caso, tendríamos que escribir:

`System.Console.WriteLine("¡Hola Mundo!");`

`class Program`

Esta línea es la definición de la clase `Program`. En una aplicación Visual C#, todas las secciones de código deben estar contenidas en una clase. La primera y la última llaves del archivo delimitan esta clase.

`static void Main()`

Esta línea define la función principal de nuestra aplicación. También se denomina **punto de entrada de la aplicación**.

Respeta la norma de las aplicaciones Visual C# que precisa que el punto de entrada de un programa es una función estática cuyo nombre es `Main` y cuyo tipo de retorno debe ser `int` o `void`. Esta función también puede recibir parámetros que se corresponden con los argumentos que se pasan por línea de comandos. La segunda y tercera llaves del archivo delimitan el código de esta función.

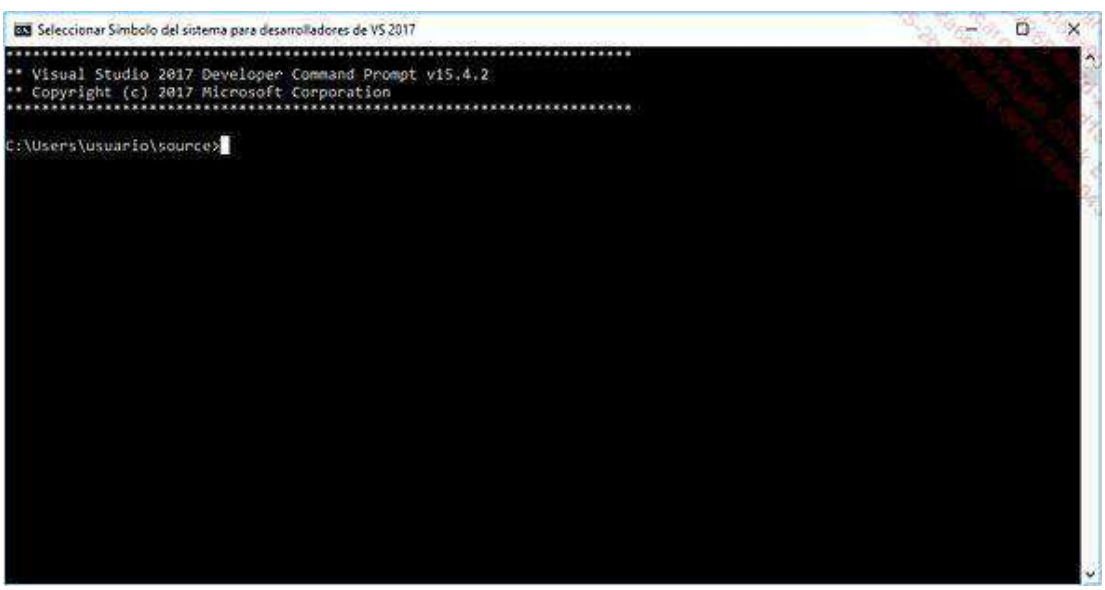
```
Console.WriteLine("¡Hola Mundo!");
```

El tipo `Console` permite manipular la consola de Windows. Posee, entre otros elementos, funciones cuyo objetivo es mostrar información o leer datos en la ventana de línea de comandos. Esta última línea se encarga de representar por pantalla el texto "¡Hola Mundo!" utilizando la función `WriteLine` de la clase `Console`.

2. Compilación

El Kit de desarrollo de software para el framework .NET (o SDK .NET) incluye una serie de herramientas que permiten, entre otras tareas, compilar las aplicaciones escritas con Visual C#. El compilador correspondiente al framework .NET 4.7 es un ejecutable llamado csc.exe que se encuentra en la carpeta C:\Windows\Microsoft.NET\Framework\v4.0.30319. La ejecución directa de este programa por línea de comandos de Windows no se recomienda, pues es necesario tener precargadas ciertas variables de entorno que pueden no estar definidas, lo cual produce un comportamiento no deseado, o incluso errores.

La instalación de Visual Studio crea con éxito un acceso directo muy útil en el menú de **Inicio** de Windows (o en la pantalla de inicio si utiliza Windows 8.1), llamado **Símbolo del sistema para desarrolladores de VS2017**. Apunta a un archivo .bat que ejecuta las instrucciones necesarias para configurar las variables de entorno indispensables para el correcto funcionamiento del compilador y, a continuación, abre una ventana de línea de comandos.



La ejecución de la instrucción `csc /?` en esta línea de comandos informa de las opciones que se pueden utilizar para la compilación. Las principales son las siguientes:

`/out:<nombre de archivo>`

Especifica el nombre del archivo de salida. Por defecto, su nombre se corresponde con el nombre del primer archivo indicado.

`/target:exe o /t:exe`

Define que el archivo de salida es una aplicación de consola.

`/target:winexe o /t:winexe`

Define que el archivo de salida es una aplicación de ventanas para Windows.

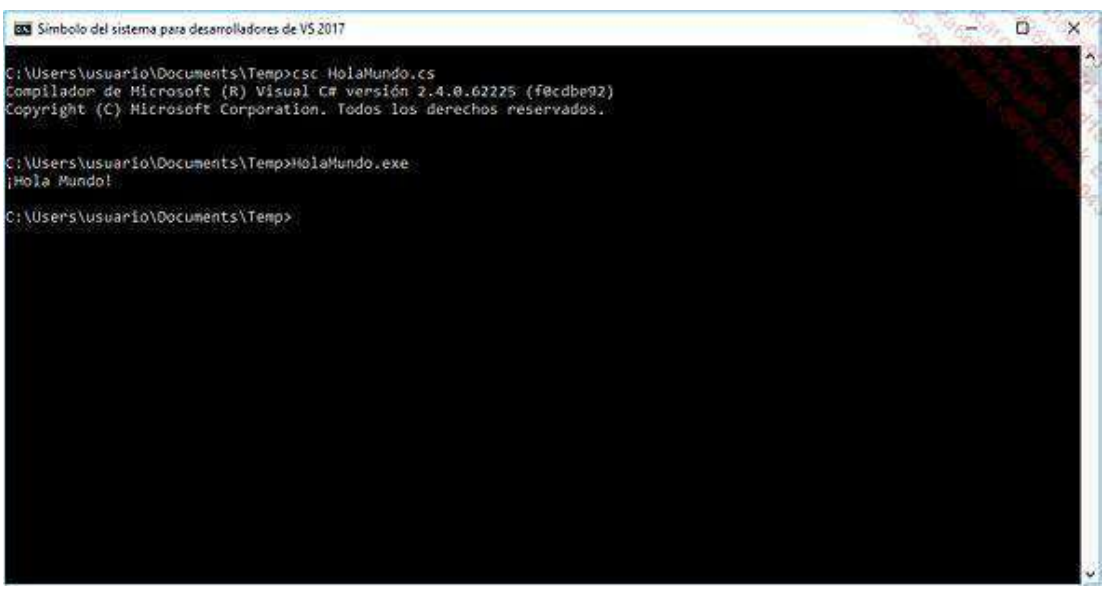
`/target:library o /t:library`

Define que el archivo de salida es una biblioteca de clases con formato DLL.

`/reference:<lista de archivos>`

Esta opción indica una lista de archivos a los que hacer referencia para la correcta compilación del código. Los nombres de los archivos deben estar separados por comas.

La compilación de la aplicación puede realizarse situando el símbolo del sistema en la carpeta que contiene nuestro archivo de código fuente y ejecutando, a continuación, el comando `csc HolaMundo.cs`. Pasados algunos instantes, el compilador toma el control y es posible comprobar la presencia del archivo ejecutable generado. Por defecto, se encuentra en la misma carpeta que el archivo de código fuente y su nombre es HolaMundo.exe.



3. Análisis del ensamblado

Una vez generado el archivo ejecutable, es posible explorarlo para analizar su contenido. Es conveniente disponer de una explicación acerca de la estructura de un archivo de ensamblado (se trata del nombre que se da a los archivos .dll o .exe construidos mediante la plataforma .NET).

a. Estructura

Los ensamblados .NET tienen la capacidad de reflectividad. Esto significa que son capaces de descubrir el contenido de un ensamblado .NET. Esto se aplica a ellos mismos: son capaces de leer su propio contenido. Esta funcionalidad existe gracias a la estructura de los ensamblados generados tras la compilación de un código fuente compatible con .NET.

Un ensamblado es una unidad lógica de código que contiene dos elementos esenciales: los **metadatos** y el código de la aplicación en forma de **código MSIL**. Los metadatos contienen información relativa al ensamblado como la

descripción de su contenido o la lista de sus dependencias. El ensamblado puede contener, también, recursos integrados, almacenados en formato binario, lo que permite no tener que disociar el código de sus dependencias externas (imágenes, archivos de sonido o de vídeo, etc.).

b. Exploración con ILDASM

Un ensamblado, al ser autodescriptivo, permite explorar sus metadatos para obtener la descripción del código que contiene y, a continuación, leer el código MSIL del ensamblado para encontrar el contenido de cada clase, función, propiedad o cualquier otro elemento del código. A esta operación se la denomina **desensamblado**.

Entre las herramientas instaladas con el kit de desarrollo del framework .NET, encontramos una herramienta de ventana llamada ILDASM (del inglés, *Intermediate Language Disassembler*). Esta herramienta ejecuta exactamente las operaciones descritas más arriba, lo que permite visualizar el código de una aplicación .NET bajo la forma de código MSIL.

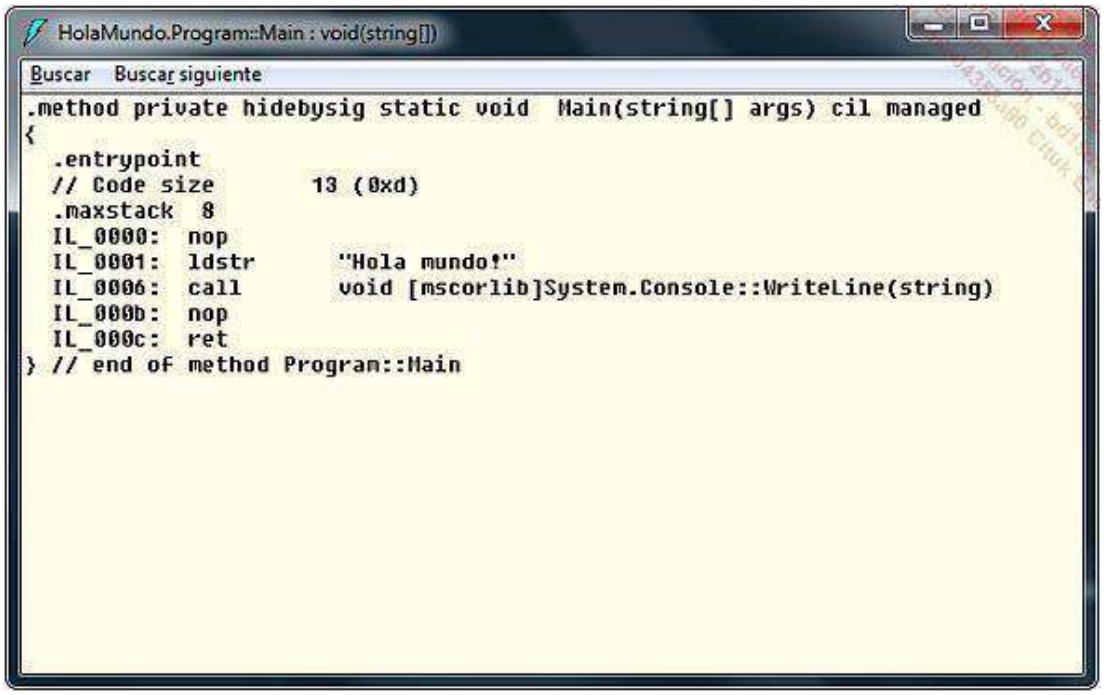
Se ejecuta mediante el comando `ildasm` desde la línea de comandos de Visual Studio. Una vez abierta la ventana principal de la aplicación, conviene proveer un ensamblado.NET para analizar mediante el menú **Archivo - Abrir**. Si abrimos el ejecutable HolaMundo.exe obtendremos el siguiente resultado:

El contenido del ensamblado se representa mediante un árbol. En este árbol encontramos los dos elementos esenciales descritos a continuación:

La sección **MANIFEST** representa los metadatos del ensamblado. Haciendo doble clic sobre este elemento se abre una ventana que contiene información relativa a su descripción.

El primer bloque indica que la versión 4.0.0.0 del ensamblado `mscorlib` es una dependencia de nuestra aplicación. Los siguientes elementos describen el ensamblado.

La segunda parte del árbol se corresponde con el código MSIL del programa. Muestra los distintos tipos definidos en el código así como su contenido. Haciendo doble clic sobre cada uno de estos elementos se abre una ventana que contiene el código asociado.



Es posible establecer una correspondencia de manera muy evidente entre el código MSIL asociado a la función `Main` del programa HolaMundo.exe y el código Visual C# escrito anteriormente.

La instrucción `IL_0001` carga la cadena de caracteres en la pila del sistema y la instrucción `IL_0006` invoca a la función `System.Console.WriteLine` almacenada en la librería `mscorlib`. El parámetro esperado por esta función se recupera desde la pila.

Instalación y primera ejecución

La primera etapa en el uso de Visual Studio es su instalación en la máquina destinada al desarrollo.

1. Requisitos previos

Visual Studio es un entorno de desarrollo que puede resultar extremadamente potente, pero para sacar el máximo partido de sus funcionalidades es preciso que la máquina sobre la que está instalado disponga de una mínima configuración de hardware y software.

Microsoft indica los siguientes valores para esa configuración:

Sistema operativo	Windows 7 SP1 Windows 8.1 (con Update 2919355) Windows Server 2012 R2 (con Update 2919355) Windows 10 (version 1507 y +) Windows Server 2016
Procesador	Procesador de 1,8 GHz
Memoria RAM	2 GB de RAM (2,5 GB en caso de ejecución sobre una máquina virtual)
Espacio en disco disponible	De 1 a 40 GB, en función de las funcionalidades instaladas
Vídeo	Tarjeta de vídeo que soporte una resolución de 1280 x 768

2. Ediciones de Visual Studio

Visual Studio 2017 está disponible en varias ediciones adaptadas a distintos contextos. Microsoft permite su descarga en la siguiente dirección: https://www.visualstudio.com/es/downloads/

Las ediciones Professional y Enterprise solo están disponibles para su evaluación durante 90 días. Pasado este periodo, será necesario obtener una licencia de uso de Microsoft.

a. Visual Studio Community

Esta versión del producto permite desarrollar aplicaciones web, Windows Store y Windows "clásicas". Ofrece, también, soporte para el desarrollo de aplicaciones multiplataforma con Xamarin.

La edición Community es suministrada Microsoft de manera gratuita bajo ciertas condiciones. Los equipos que trabajen en proyectos open source o investigación universitaria pueden, en particular, acceder a ella gratuitamente.

En general, cualquier persona puede utilizar la herramienta sin coste alguno, siempre con fines educativos o de formación.

Visual Studio 2017 Community puede ser utilizada también para fines comerciales por desarrolladores independientes o pequeños equipos profesionales (como máximo 5 usuarios y empresas de menos de 250 empleados y menos de un millón de dólares de volumen anual de negocio).

Esta versión de Visual Studio es la que se utiliza a lo largo de este libro. Puede utilizarla para compilar y ejecutar los ejemplos de código que encontrará a lo largo del libro.

b. Ediciones comerciales

Visual Studio 2017 se presenta en tres versiones comerciales diferentes adaptadas a problemáticas muy variadas.

Visual Studio Test Professional

Esta versión está destinada a aquellos profesionales encargados de probar aplicaciones. No permite desarrollar y, por lo tanto, no se puede utilizar en el ámbito de este libro.

Visual Studio Professional

Esta versión del producto ofrece básicamente las mismas funcionalidades que la edición Community, pero está adaptada a un uso corporativo (más de 5 usuarios, más de 250 empleados, o un volumen de negocio anual superior a un millón de dólares). La licencia para esta edición tiene diferentes servicios (formación, soporte, herramientas).

Visual Studio Enterprise

Esta última edición de Visual Studio integra las funcionalidades de la edición Professional e incluye funcionalidades avanzadas tales como la automatización de pruebas de interfaz, la búsqueda de código duplicado o el análisis de la arquitectura de una aplicación. Incluye, también, soluciones punteras para el diagnóstico mediante datos que provienen de aplicaciones en producción, así como herramientas que permiten realizar pruebas de carga de aplicaciones web. La lista de servicios y herramientas complementarias proporcionadas con la licencia de uso se hace más grande, sobre todo con una licencia Office Pro Plus y una cuenta de desarrollador Windows Store.

3. Instalación

Además de respetar los requisitos previos enumerados al principio de este capítulo, la instalación de Visual Studio requiere un soporte de instalación que puede encontrar en http://www.visualstudio.com.

Cabe destacar que Microsoft no proporciona ninguna imagen de disco (.iso) para esta versión de Visual Studio, lo cual puede resultar particularmente problemático en el contexto de una instalación sin conexión o cuando se tienen que llevar a cabo varias instalaciones. El programa de instalación ha sufrido una revisión completa para esta nueva versión: muy modular, permite iniciar un proceso de instalación "conectado", con la descarga sobre la marcha de cada elemento necesario, pero también da la posibilidad de crear imágenes de instalación personalizadas, sin elementos que no se vayan a utilizar. Microsoft ha creado un procedimiento, que se explica a continuación, para realizar esta operación.

En ambos casos, en primer lugar es indispensable descargar el programa de arranque de Visual Studio de https://www.visualstudio.com/es/downloads/. Es un programa muy ligero que sirve como puerta de entrada para el resto de operaciones. Si se ejecuta normalmente, este programa muestra una interfaz gráfica dedicada a la personalización de la instalación. Se detallará más adelante.

La instalación fuera de línea es ligeramente más compleja porque necesita algunas acciones previas. El principio básico utilizado para la creación de una imagen de instalación es muy simple: el programa de arranque descarga una lista definida de paquetes para construir una caché local. Esta último se puede utilizar durante la ejecución del proceso de instalación

El proceso de descarga lo inicia la ejecución del programa de arranque con una línea de comando. Para descargar los paquetes que permiten instalar todas las funcionalidades de Visual Studio, escriba el siguiente comando:

```
vs_community.exe --layout <carpeta de destino> --lang es-ES
```

De todos modos, si quiere limitar la descarga solo a las funcionalidades que va a necesitar, puede especificar qué componentes incluir a través de los parámetros `--add`, `--includeRecommended` e `--includeOptional`. Los componentes se agrupan por "cargas de trabajo" (*workloads*), lo que permite integrar rápidamente las principales funcionalidades. Para descargar solo los paquetes relativos al desarrollo de aplicaciones de escritorio, el comando es:

```
vs_community.exe --layout <carpeta de destino> --add
Microsoft.VisualStudio.Workload.ManagedDesktop
--includeRecommended --includeOptional --lang es-ES
```

Esta configuración es suficiente para la ejecución de los ejemplos incluidos en este libro.

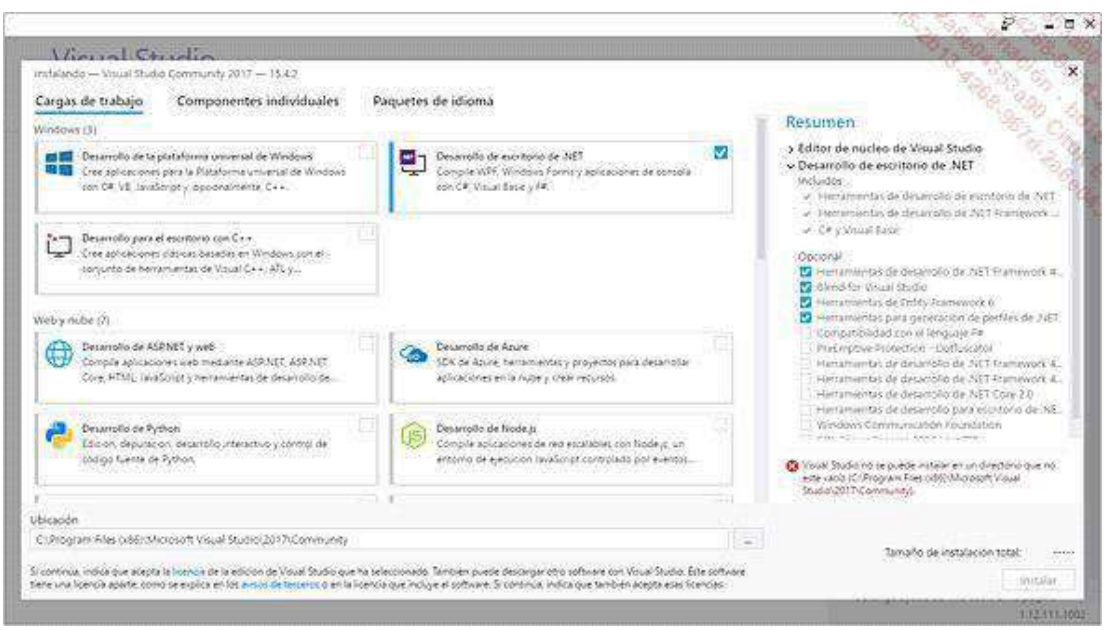
La lista completa de las cargas de trabajo y los componentes de Visual Studio Community está disponible en la siguiente dirección: https://docs.microsoft.com/es-es/visualstudio/install/workload-component-id-vs-community

La documentación completa relativa al uso del programa de inicio por línea de comandos la puede encontrar en: https://docs.microsoft.com/es-es/visualstudio/install/use-command-line-parameters-to-install-visual-studio

Después del proceso de descarga, cuya duración depende de los componentes elegidos y de la velocidad de la conexión a Internet, la carpeta de destino contiene un gran número de carpetas así como un ejecutable (vs_setup.exe) que se debe ejecutar para iniciar la instalación. Para un almacenamiento más sencillo, se puede utilizar una herramienta externa de creación de imágenes de disco (IsoCreator, por ejemplo) que genera un archivo .iso.

En este momento, dispone de un programa de arranque o de una imagen de instalación que corresponde a sus necesidades. Puede comenzar la instalación del entorno de desarrollo.

Una vez se ha ejecutado el programa de inicio (vs_community.exe) o el programa de instalación (vs_setup.exe), se muestra la siguiente pantalla.



Esta interfaz gráfica tiene como objetivo principal permitir la selección de las funcionalidades que se deben instalar. Come se decía anteriormente, los componentes se agrupan en conjuntos, las cargas de trabajo, que representan los diferentes tipos de desarrollo que se pueden realizar:

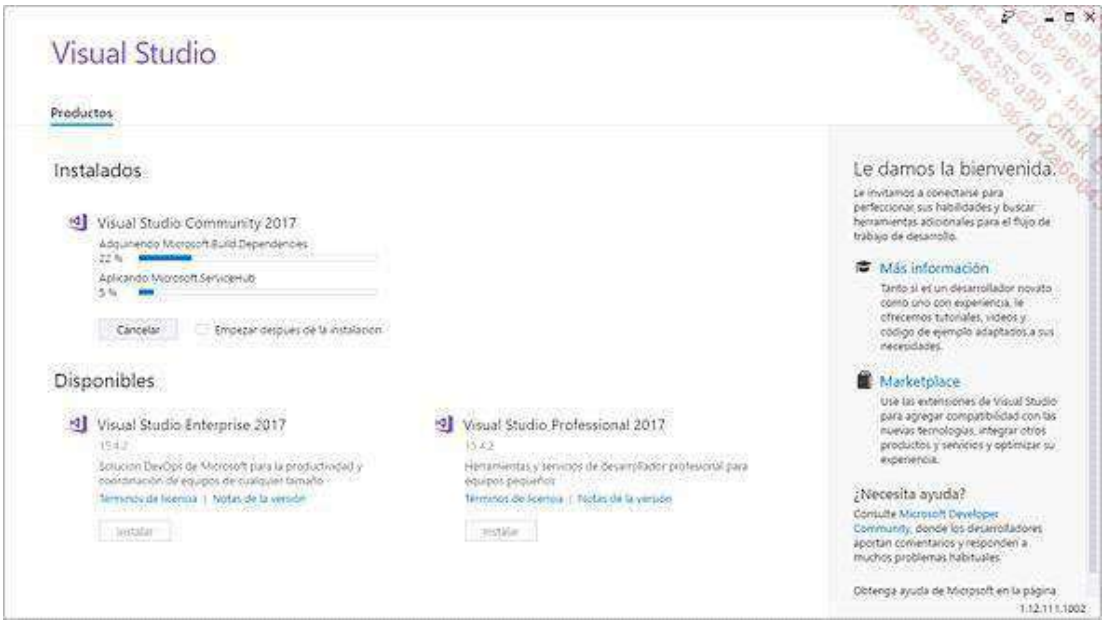
- Desarrollo para la plataforma Windows universal
- Desarrollo .NET Desktop
- Desarrollo Desktop en C++
- Desarrollo web y ASP.NET
- Desarrollo Azure
- Desarrollo Python
- Desarrollo Node.js
- Almacenamiento y tratamiento de datos
- Aplicaciones científicas y análisis de datos
- Desarrollo Office/SharePoint
- Desarrollo móvil en .NET
- Desarrollo de juegos con Unity
- Desarrollo móvil en JavaScript
- Desarrollo móvil en C++
- Desarrollo de juegos en C++
- Desarrollo de extensiones Visual Studio
- Desarrollo Linux en C++
- Desarrollo multiplataforma .NET Core

Cada una de estas cargas de trabajo se puede adaptar a las necesidades específicas del desarrollador: para cada categoría, la parte derecha de la interfaz muestra los componentes opcionales asociados y permite su selección o deselección. Es posible mostrar información que explica el interés de algún elemento de la lista situando el cursor del ratón sobre el nombre de la funcionalidad.

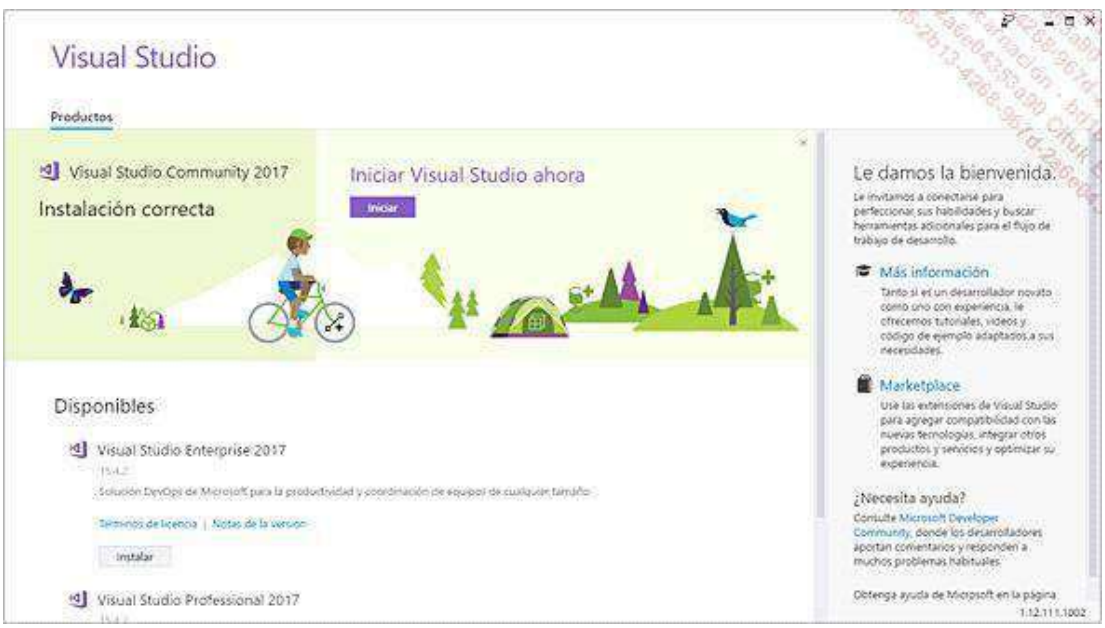
Esta pantalla también permite modificar la carpeta de instalación del producto e informa que la validación de la instalación implica la aceptación del contrato de licencia. La última indicación importante que se muestra en esta pantalla es la estimación del espacio en disco necesario para la configuración elegida. Evidentemente, cambia con cada modificación de la selección.

Llegados a este punto, basta con hacer clic en el botón **Instalar** para iniciar el proceso de instalación. Este puede resultar bastante largo (¡varias horas!) según el número de funcionalidades seleccionadas y los elementos que estén en la caché de su máquina. De hecho, en el caso de una instalación desde el programa de inicio, se debe descargar y descomprimir cada componente antes de colocarlo en su sitio definitivo. La velocidad de la conexión a Internet puede ser un factor importante, así como la potencia de la máquina.

Una vez iniciado el proceso de instalación, la interfaz muestra una nueva ventana que indica el grado de avance.



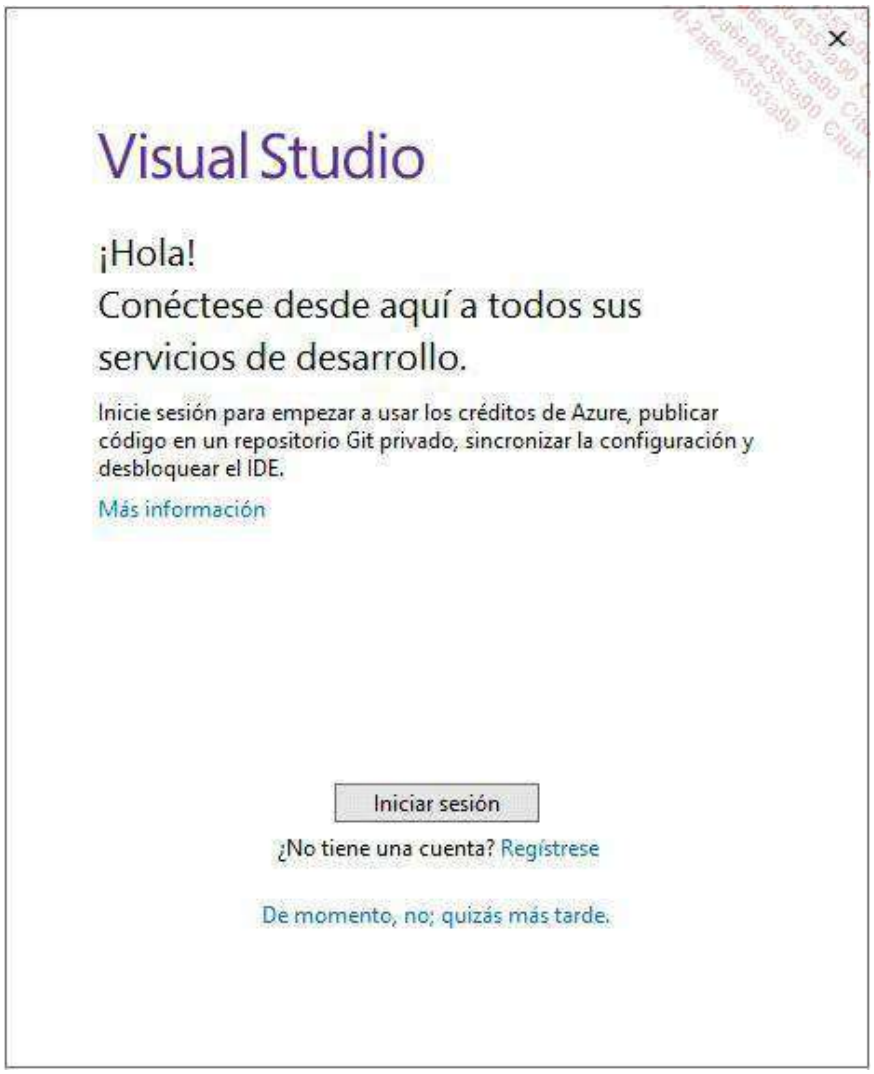
Una vez terminada la instalación, aparece una ventana que le informa de que la operación ha finalizado con éxito y le propone el reinicio de la máquina (que, evidentemente, debe aceptarse).



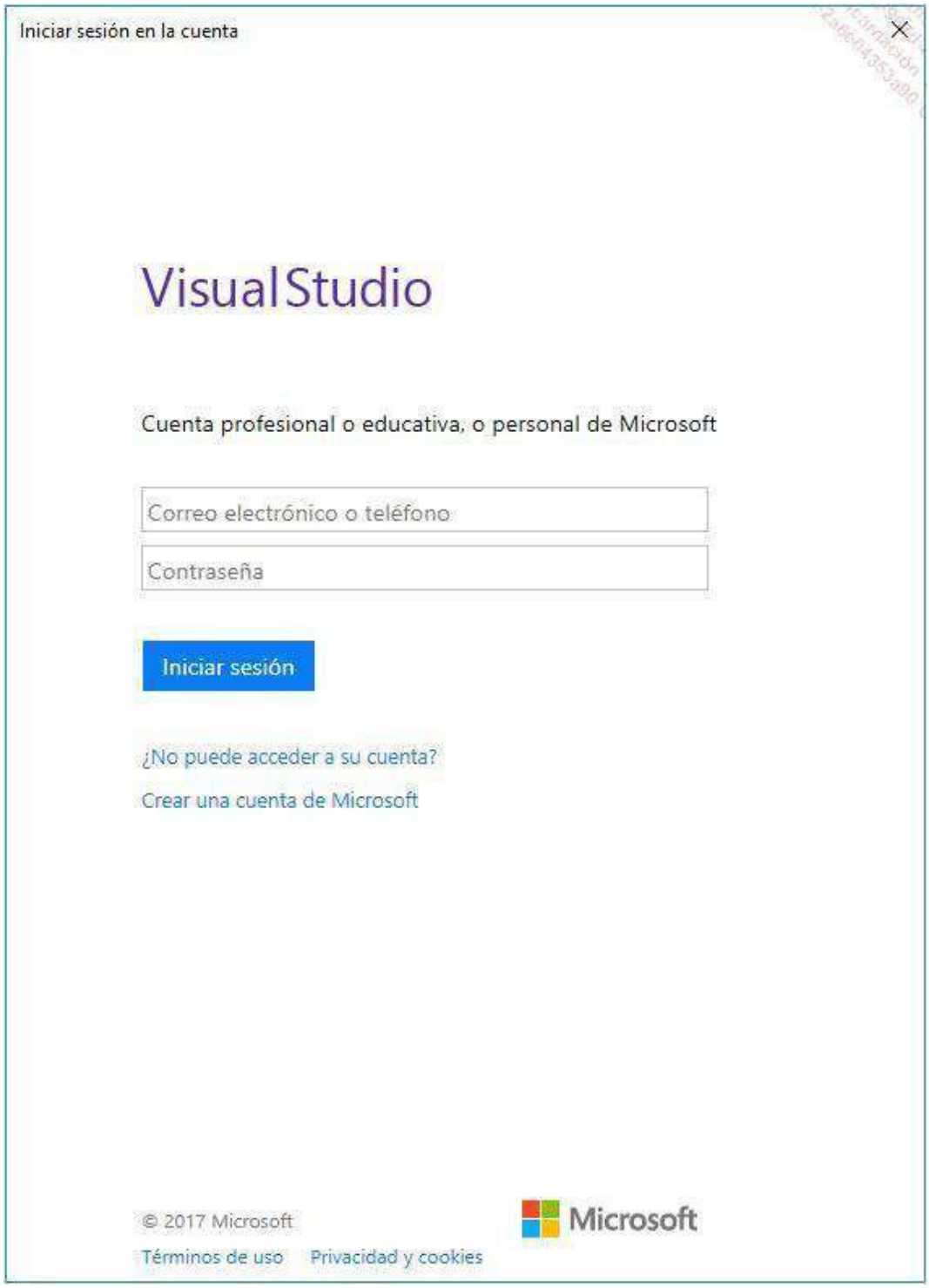
4. Primera ejecución

Durante la instalación del producto, automáticamente en el menú de **Inicio** (o en el Start Screen, si utiliza Windows 8 u 8.1) se crea un acceso directo que le permite ejecutar Visual Studio.

La aplicación aprovecha la primera ejecución para proponerle personalizar su entorno de desarrollo. La primera etapa de esta personalización es la conexión a una cuenta Microsoft para importar y salvaguardar sus parámetros de Visual Studio.



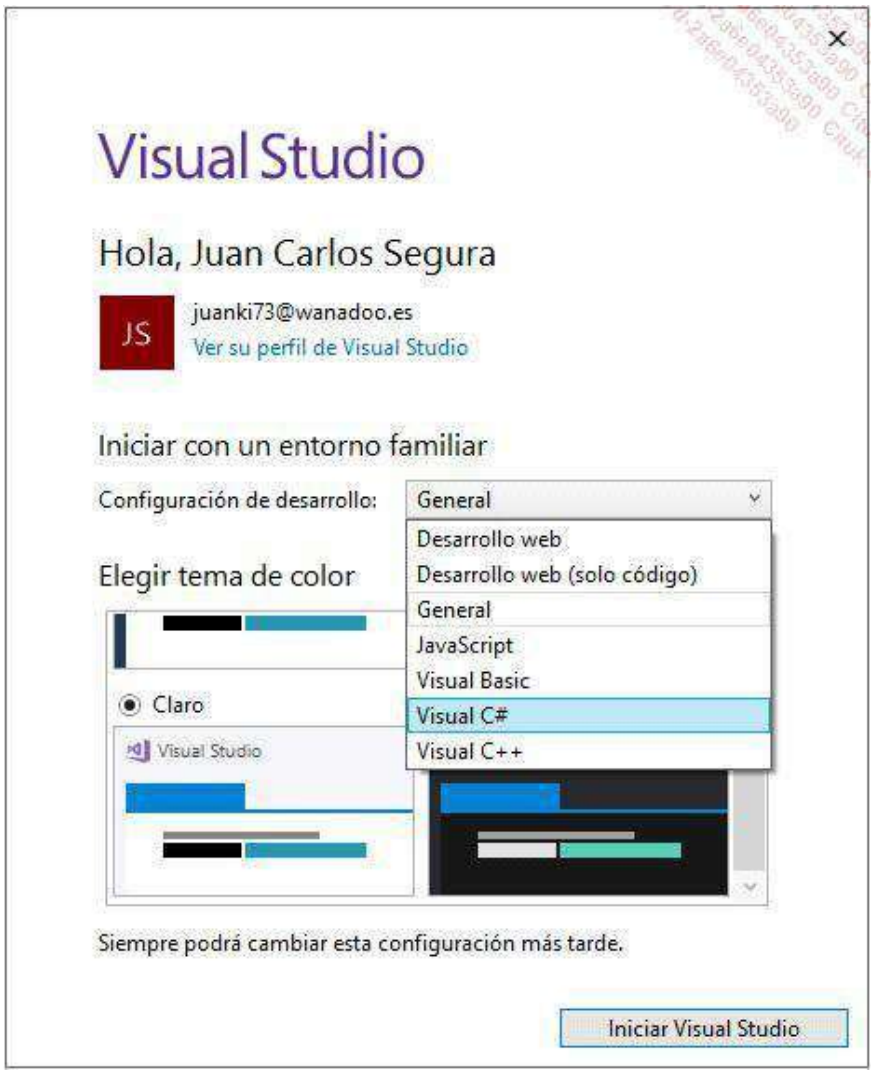
Haga clic en el botón **Iniciar sesión** para abrir una ventana de conexión a una cuenta Microsoft que le permita acceder a su cuenta de buzón de correo de Microsoft (en la dirección http://www.outlook.com o http://www.hotmail.com si es un nostálgico).



Una vez establecida la conexión, Visual Studio puede solicitar información suplementaria para personalizar un poco su interfaz y le propone, a su vez, la creación de una cuenta de **Visual Studio Online**. Este servicio se corresponde con una versión ligera y en línea del **Team Foundation Server**. Incluye la gestión del código mediante el control de código fuente e incluye también la planificación de los elementos de trabajo a lo largo del ciclo de vida de un proyecto.

Una vez superada esta etapa, Visual Studio le propone que seleccione un tema para el entorno: **Azul**, **Claro** u **Oscuro**. Escoja el que más le guste.

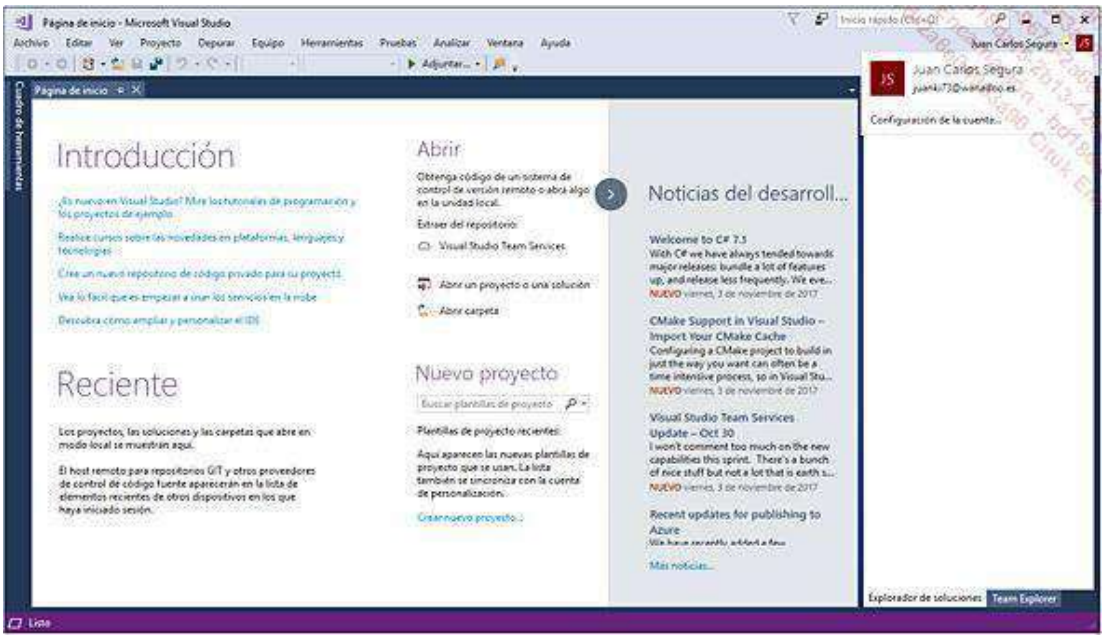
En la primera pantalla se muestra una lista desplegable que presenta una lista de parámetros de desarrollo predefinidos. Los ejemplos y explicaciones de este libro se han realizado con una configuración de desarrollo **Visual C#**.



Por último, tras hacer clic en el botón **Iniciar Visual Studio** y esperar algunos instantes durante los que se lleva a cabo la configuración del entorno, aparece la página de inicio de Visual Studio.

Esta página permite acceder rápidamente a los últimos proyectos abiertos así como crear o abrir un proyecto nuevo. Muestra, también, cierta información relativa a los productos de desarrollo de Microsoft.

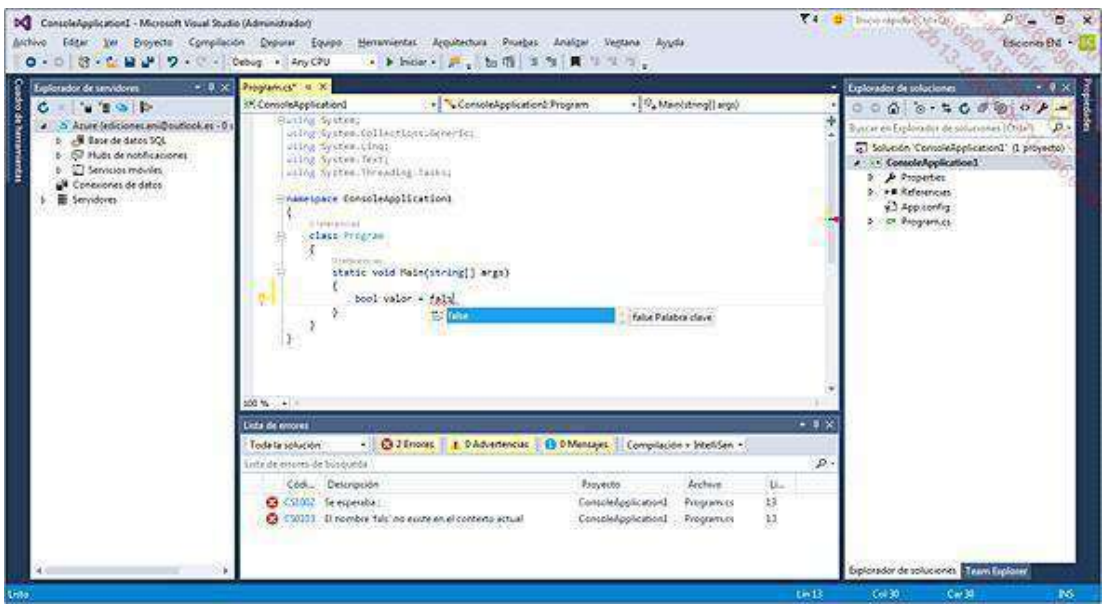
Un indicador situado en la zona superior derecha le permite conocer qué cuenta de Microsoft está conectada a Visual Studio.


Página de inicio - Microsoft Visual Studio
Archivo Editar Ver Proyecto Depurar Equipo Herramientas Pruebas Analizar Ventana Ayuda
Inicio rápido (Ctrl+Q)
Juan Carlos Segura
Adjuntar...
Cuadro de herramientas
Página de inicio
Introducción
¿Es nuevo en Visual Studio? Mire los tutoriales de programación y los proyectos de ejemplo.
Realice cursos sobre las novedades en plataformas, lenguajes y tecnologías.
Cree un nuevo repositorio de código privado para su proyecto.
Vea lo fácil que es empezar a usar los servicios en la nube.
Descubra cómo ampliar y personalizar el IDE.
Reciente
Los proyectos, las soluciones y las carpetas que abre en modo local se mostrarán aquí.
El host remoto para repositorios GIT y otros proveedores de control de código fuente aparecerán en la lista de elementos recientes de otros dispositivos en los que haya iniciado sesión.
Abrir
Obtenga código de un sistema de control de versión remoto o abra algo en la unidad local.
Extraer del repositorio:
Visual Studio Team Services
Abrir un proyecto o una solución
Abrir carpeta
Nuevo proyecto
Buscar plantillas de proyecto
Plantillas de proyecto recientes:
Aquí aparecen las nuevas plantillas de proyecto que se usan. La lista también se sincroniza con la cuenta de personalización.
Crear nuevo proyecto...
Noticias del desarroll...
Welcome to C# 7.1
With C# we have always tended towards major releases: bundle a lot of features up, and release less frequently. We eve...
NUEVO viernes, 3 de noviembre de 2017
CMake Support in Visual Studio – Import Your CMake Cache
Configuring a CMake project to build in just the way you want can often be a time intensive process, so in Visual Stu...
NUEVO viernes, 3 de noviembre de 2017
Visual Studio Team Services Update – Oct 30
I won't comment too much on the new capabilities this sprint. There's a bunch of nice stuff but not a lot that is earth s...
NUEVO viernes, 3 de noviembre de 2017
Recent updates for publishing to Azure
Más noticias...
Juan Carlos Segura
Configuración de la cuenta...
Explorador de soluciones
Team Explorer
Lista

Descripción de las herramientas

Visual Studio está constituido por tres tipos de elementos que se reparten en su zona de representación:

- La parte superior de la ventana principal está compuesta por una barra de menús y varias barras de herramientas.
- La parte central de la ventana principal contiene la zona de trabajo en la que es posible visualizar, entre otros, uno o varios diseñadores visuales así como ventanas de edición de código.
- Varias ventanas secundarias representan las distintas herramientas a disposición del desarrollador.

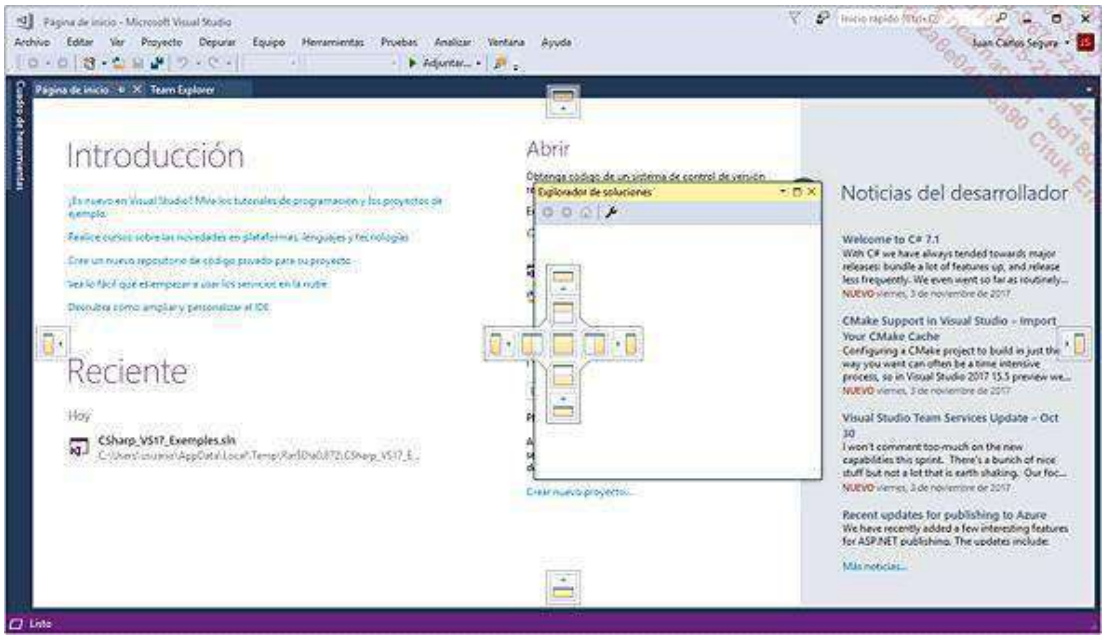


El entorno de trabajo puede gestionarse y adaptarse mediante el uso de tres técnicas:

- El anclado de ventanas,
- La ocultación automática de ventanas,
- El uso de pestañas.

Las ventanas pueden anclarse a uno de los lados de la ventana principal. También es posible hacer que una ventana sea flotante, es decir, que esté posicionada de manera absoluta, sin relación alguna con los elementos de la interfaz. Estas dos operaciones pueden llevarse a cabo con ayuda del menú contextual de cada una de las ventanas (que se abre haciendo clic con el botón derecho en la barra de título de la ventana) o arrastrando y soltando.

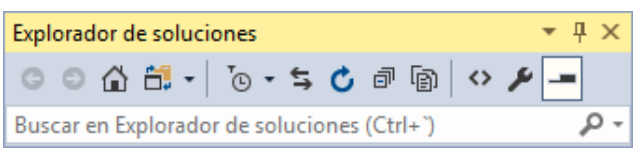
Una vez se ha desplazado una ventana arrastrando y soltando, Visual Studio muestra varias guías. Cuando la ubicación de la ventana se corresponde con un anclaje, se previsualiza el espacio ocupado mediante un fondo de color azul (poco importa, en este punto, el tema seleccionado para Visual Studio).



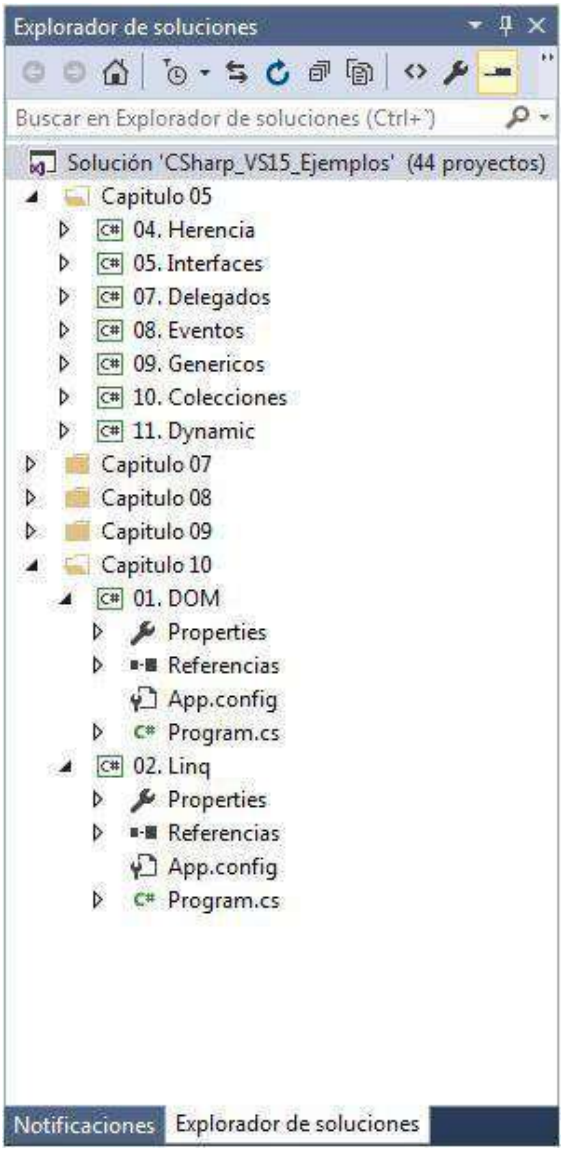
Situar una ventana sobre alguna de las guías ancla la ventana en la posición deseada. Si se desvincula de la guía, la ventana se convierte en flotante.

La ocultación automática de la ventana puede resultar efectiva sobre una ventana anclada. Cada ventana posee en su barra de título un botón que representa una chincheta. Cuando está orientada horizontalmente, la ventana está en modo de ocultación automática.

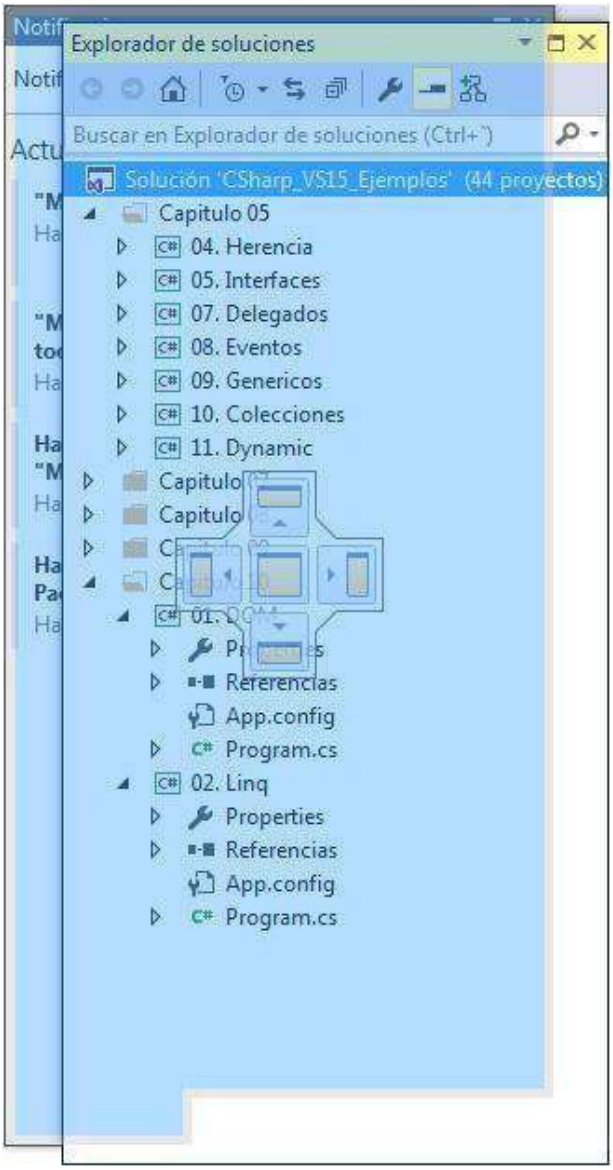
En caso contrario, está anclada a la ventana principal de Visual Studio:



El uso de pestañas permite anclar ventanas con una pérdida mínima de espacio de trabajo. Esta solución consiste en utilizar una ventana anclada para mostrar varias ventanas mediante una barra de pestañas.

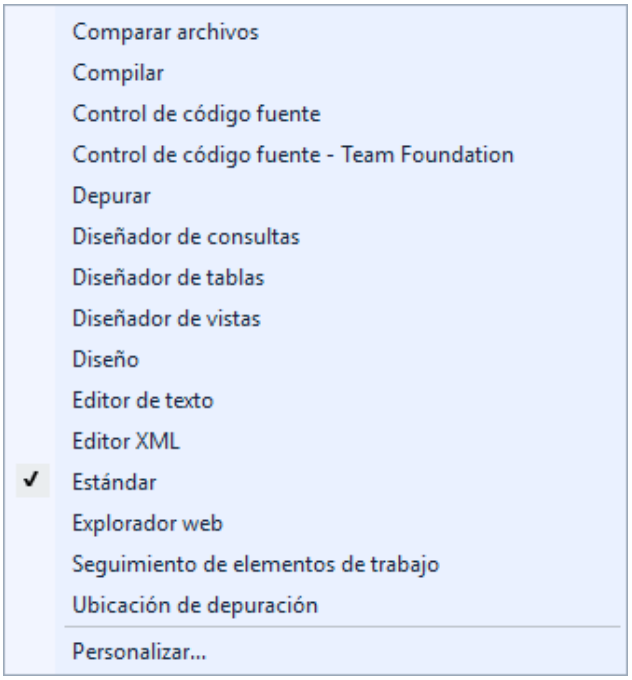


Este resultado se obtiene situando una ventana sobre otra mediante las guías que se muestran en Visual Studio o sobre la barra de título de la ventana de destino. Visual Studio muestra una visualización previa que incluye la pestaña en la zona inferior.



1. Barras de herramientas

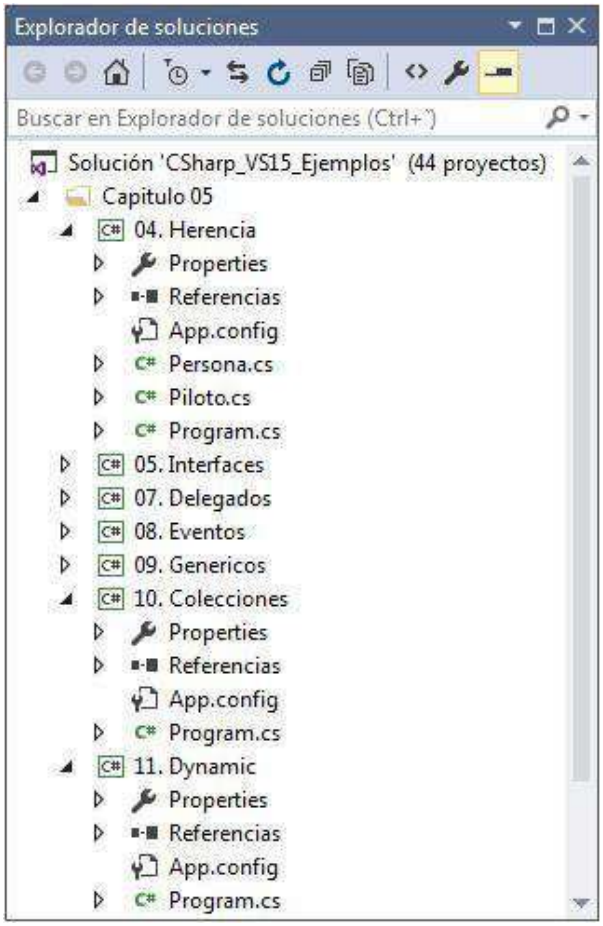
Visual Studio dispone de numerosas barras de herramientas adaptadas a distintos dominios funcionales. Es posible personalizar su representación: el menú contextual de la zona de barras de herramientas muestra cierto número de barras de herramientas disponibles.



La selección de un elemento de esta lista muestra la barra de herramientas correspondiente en la zona de barras de herramientas. Esta personalización es importante, pues ciertas herramientas pueden resultar inútiles en algunos contextos y particularmente interesantes en otros momentos. La barra de herramientas **Control de código fuente - Team Foundation**, por ejemplo, es completamente inútil si no utiliza **Team Foundation Server** o **Visual Studio Online**, mientras que resultará indispensable si utiliza una de estas dos herramientas.

2. Explorador de soluciones

El explorador de soluciones es una herramienta indispensable. Permite navegar entre las distintas carpetas y archivos que componen el código fuente de una solución. Una solución puede contener varios proyectos, que se corresponden con varias aplicaciones y/o bibliotecas de clases (y/o aplicaciones web, etc.).



El uso de esta herramienta se describe con detalle en el capítulo La organización de una aplicación.

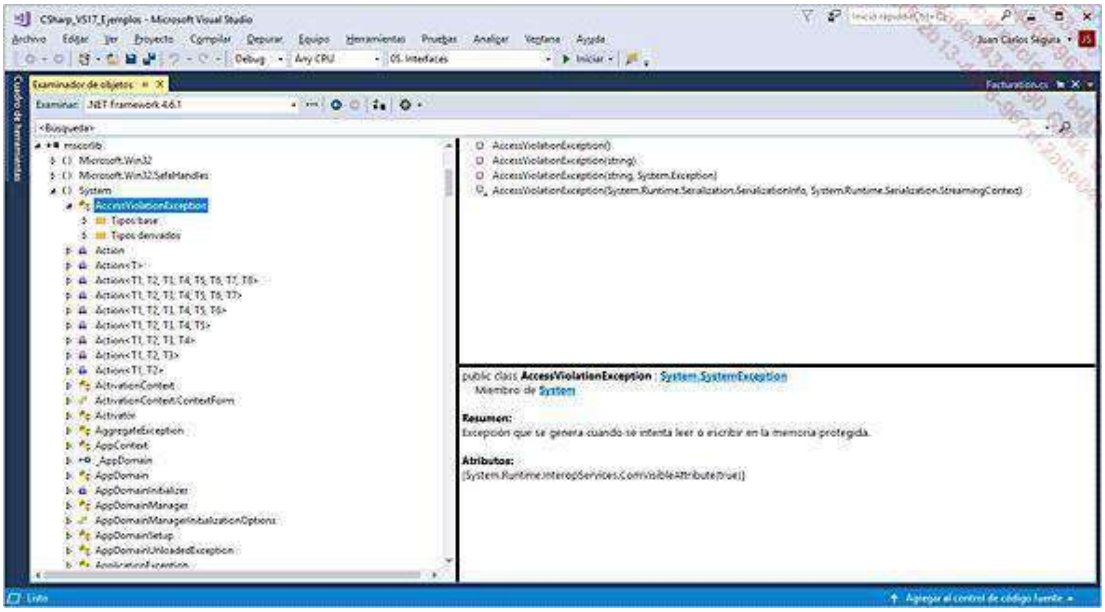
3. Examinador de objetos

El examinador de objetos permite obtener una visión detallada de dos grupos de elementos:

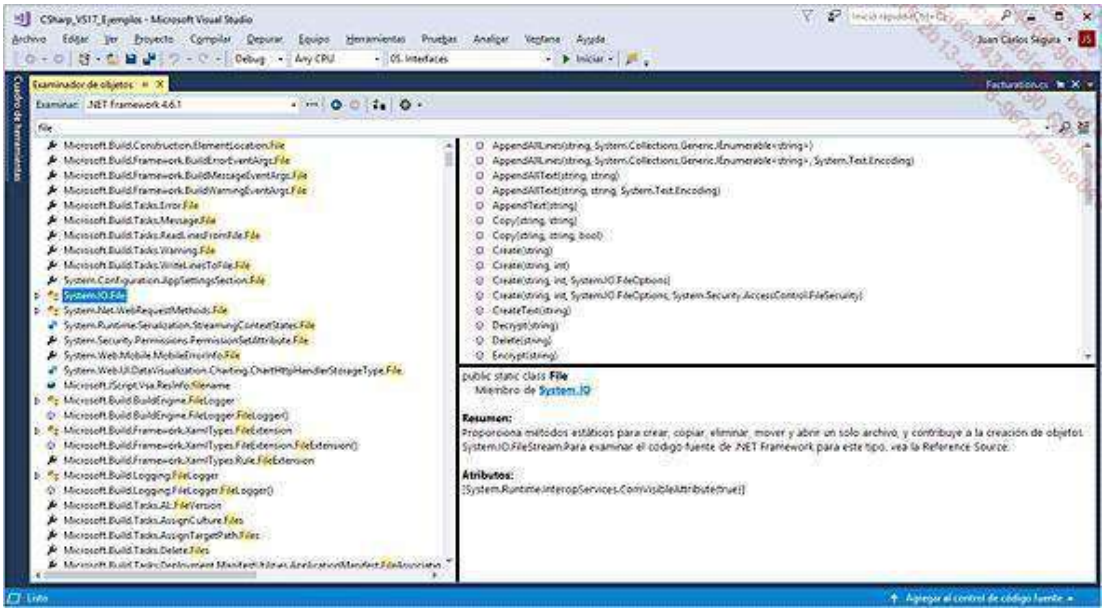
- Los proyectos, espacios de nombres y tipos que componen la solución activa,
- Los ensamblados, espacios de nombres y tipos que componen un sub-conjunto del framework .NET.

Esta herramienta puede abrirse mediante el menú **Ver - Examinador de objetos** o mediante la combinación de teclas [Ctrl] W, J.

Cuando se selecciona un elemento, su descripción se muestra en la parte derecha de la ventana. Si la selección se realiza sobre un tipo que contiene varios miembros, estos se muestran también en la parte derecha de la ventana.



Una zona para introducir texto permite realizar una búsqueda entre los elementos inspeccionados. Esta búsqueda puede devolver resultados sobre todo o parte de un nombre de clase, de un espacio de nombres o incluso de una función o una propiedad.

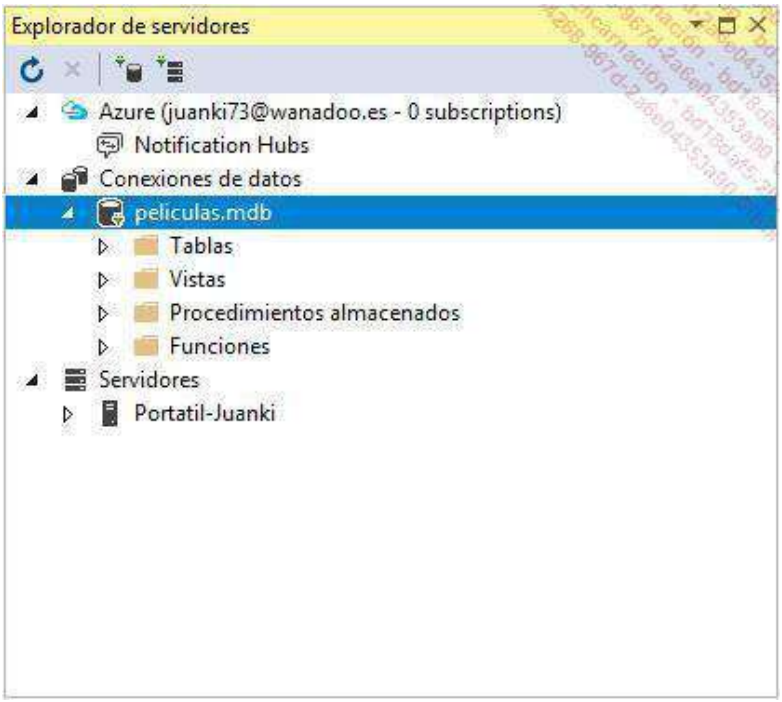


4. Explorador de servidores

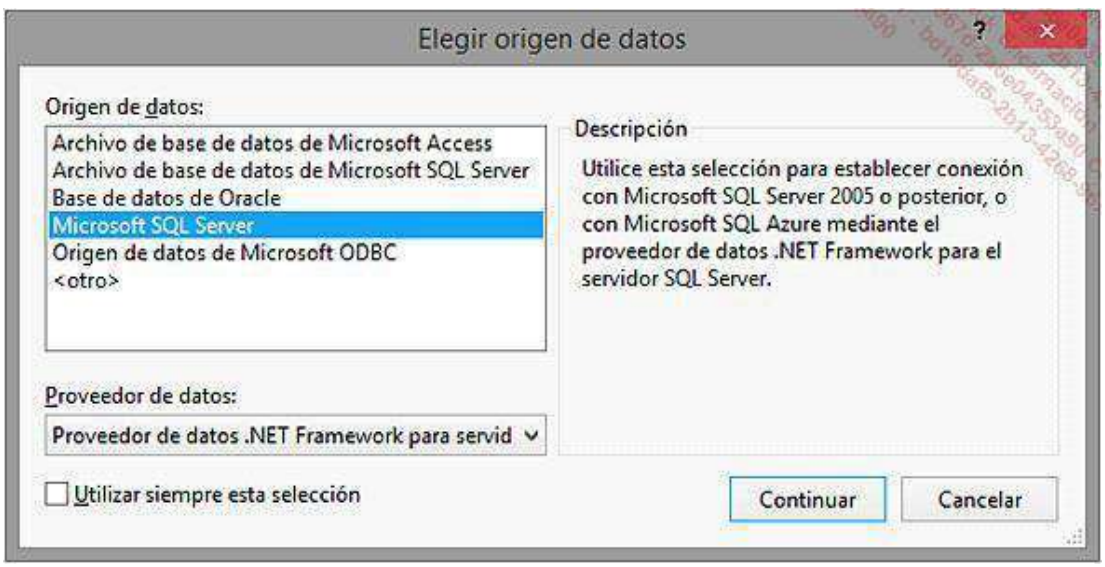
El explorador de servidores permite visualizar y manipular datos que provienen de distintas fuentes: una base de datos, indicadores de rendimiento de Windows o una cuenta de Microsoft Azure. También es posible realizar tareas de administración sobre las distintas fuentes, como crear una tabla en una base de datos o agregar un sitio web en Azure.

Esta herramienta puede abrirse mediante el menú **Ver - Explorador de servidores** o mediante la combinación de teclas [Ctrl] W, L.

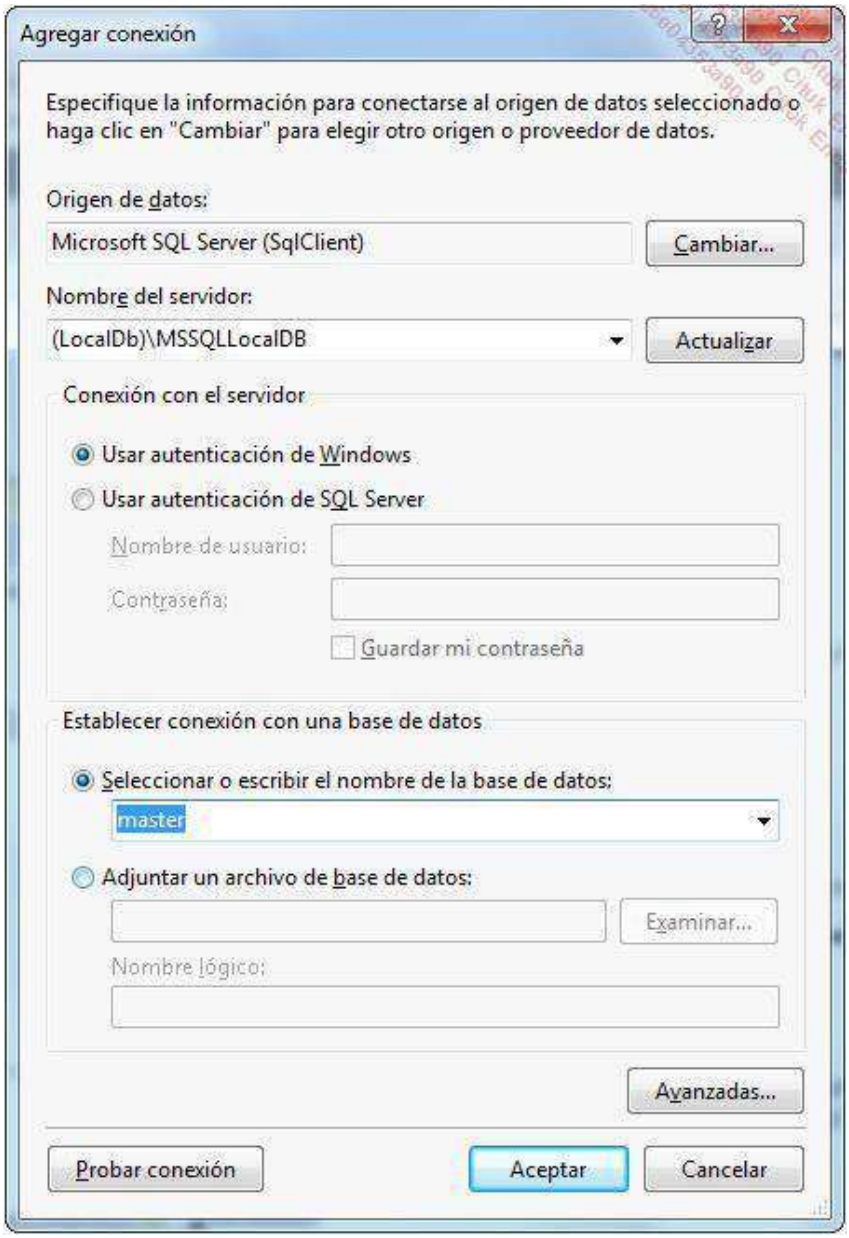
La administración de los servicios albergados en Windows Azure requiere poseer una cuenta en esta plataforma. Puede crear una cuenta de prueba en la siguiente dirección: http://azure.microsoft.com/es-es/pricing/free-trial/



La barra de herramientas de esta ventana presenta varios botones que permiten conectarse a distintas fuentes de datos. El uso del botón **Conectar con base de datos** abre una ventana que contiene una lista de proveedores de datos.



La selección de uno de estos proveedores (SQL Server en nuestro ejemplo) abre una ventana que permite introducir y/o seleccionar el nombre de un servidor de bases de datos y, a continuación, seleccionar la base de datos con la que se desea establecer una conexión.



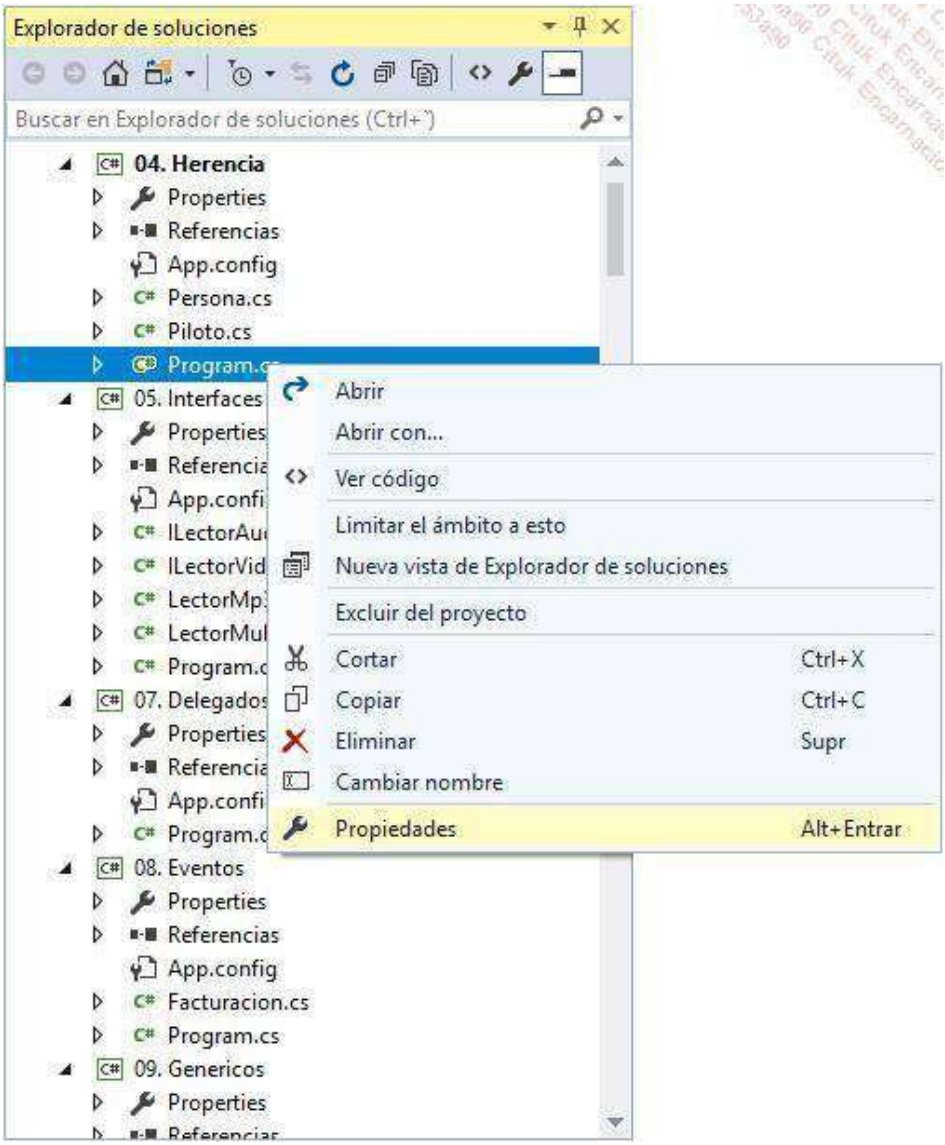
LocalDb es un modo de ejecución particular de **SQL Server Express** incluido a partir de la versión 2012. Este modo está destinado a los desarrolladores de aplicaciones. Esta edición de SQL Server (en versión 2016) es gratuita y puede descargarse del sitio web de Microsoft (en el momento de escribir estas líneas se encuentra en la siguiente dirección: https://www.microsoft.com/en-us/sql-server/sql-server-downloads).

Una vez validada la selección, se agrega una nueva conexión en el árbol del explorador de servidores.

5. Ventana de propiedades

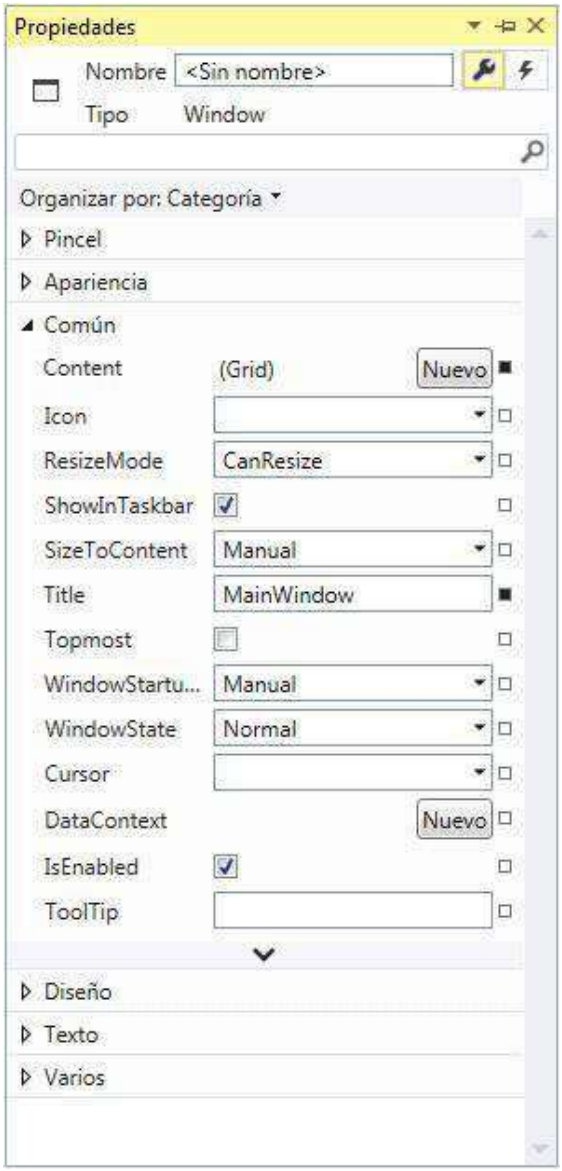
Ciertos elementos relativos a gran parte de los componentes de una solución pueden mostrarse mediante una **ventana de propiedades**. Estos componentes pueden ser objetos del explorador de soluciones o corresponderse con algún componente gráfico, cuando se está editando una ventana de aplicación, por ejemplo. También es posible mostrar las propiedades de un servidor o de una tabla de base de datos a partir del **explorador de servidores**.

La **ventana de propiedades** puede abrirse presionando la tecla [F4] o seleccionando la opción **Propiedades**, presente en el menú contextual de muchos elementos de la interfaz de Visual Studio. En concreto, esta opción está presente en el menú contextual de cada uno de los archivos contenidos en el proyecto.



La ventana **Propiedades** presenta distintos datos que se corresponden con el último elemento seleccionado. Estos elementos aparecen en forma de listado, donde la primera columna se corresponde con el nombre de la propiedad y la segunda con el valor asociado. Este listado puede ordenarse por categoría, lo que modifica ligeramente la presentación de cara a tener en cuenta una agrupación de elementos.

La siguiente imagen muestra la apariencia de la ventana durante la edición de un elemento gráfico en el editor de Visual Studio.

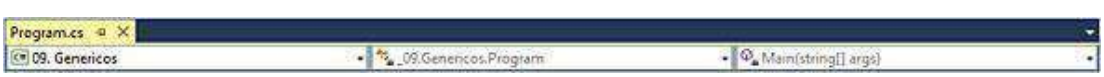


6. Ventana de edición de código

Esta ventana es sin duda la más importante del entorno de desarrollo, pues permite visualizar y modificar el código fuente de una aplicación. Integra numerosas herramientas para acelerar la lectura y escritura de código. Es posible abrir varios archivos simultáneamente, cada uno asociado con una ventana de edición diferente.

a. Navegación

La ventana de edición de código posee, además de la zona de introducción de texto, tres listas desplegables que permiten navegar rápidamente entre las distintas zonas de una solución.



La primera de estas listas desplegables se utiliza cuando se comparte un archivo de código fuente entre varios proyectos en el interior de una misma solución. Contiene la lista de proyectos asociados y permite situarse en el proyecto seleccionado.

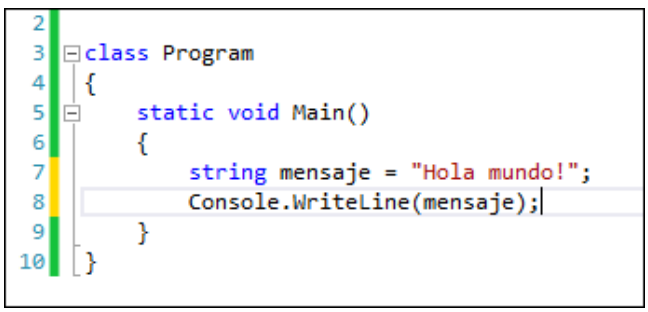
La segunda permite navegar entre las distintas definiciones de tipo contenidas en el archivo en curso.

La última lista contiene los nombres de los distintos miembros del tipo sobre el que se ha posicionado el editor de texto. La selección de uno de estos elementos desplaza la vista en curso al nivel de la definición del miembro.

b. Seguimiento de las modificaciones

Puede seguir las modificaciones de los archivos abiertos en Visual Studio. Cuando se modifica una línea se muestra un borde de color a su izquierda.

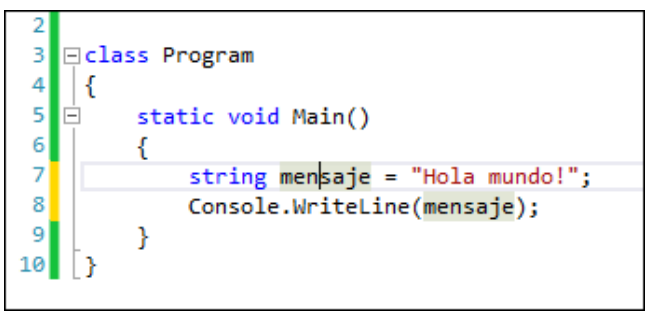
Un borde de color amarillo indica que el código se ha modificado, pero no se ha guardado todavía. Cuando el código modificado se guarda el borde se vuelve de color verde.



Es posible habilitar o deshabilitar la numeración de las líneas en el editor mediante la ventana de opciones de Visual Studio. Puede acceder a este parámetro mediante el menú **Herramientas - Opciones** y, a continuación, **Editor de texto - C#**. A continuación, marque o desmarque la opción **Números de línea**.

c. Resaltar referencias

Cuando se sitúa el cursor sobre el nombre de una variable, función o clase, cada uno de los usos de ese elemento se subraya en la ventana de edición.



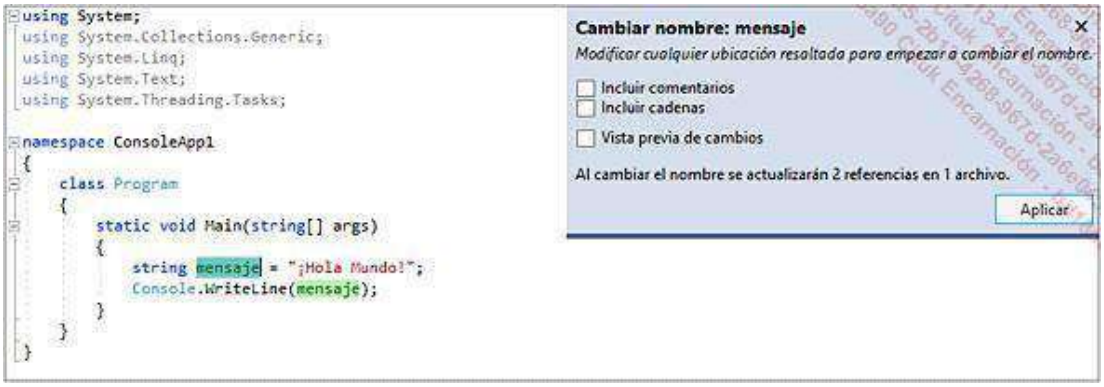
d. Refactorización

Puede modificar el nombre de un elemento del código en el editor y propagar automáticamente las modificaciones a todas las instrucciones que utilizan ese elemento. Esta funcionalidad, llamada **refactorización**, se utiliza habitualmente para modificar las ocurrencias de una variable o una función. Esta operación puede llevarse a cabo de dos maneras.

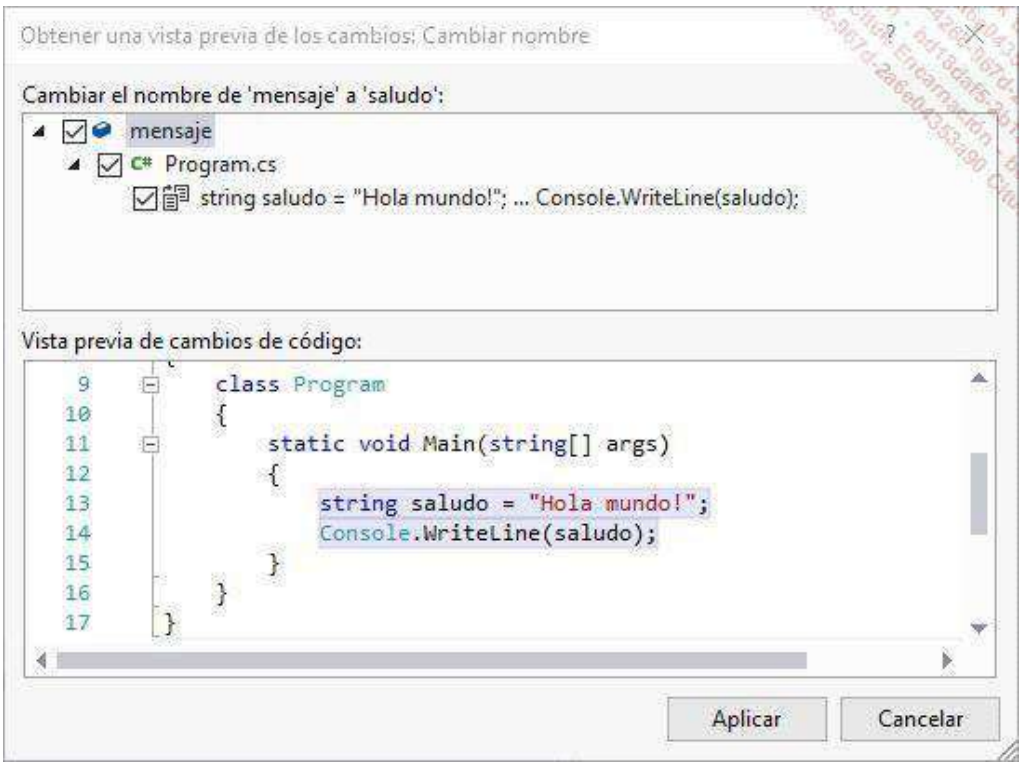
Antes de renombrar

Se selecciona la opción **Cambiar nombre** en el menú contextual del nombre del elemento que se desea renombrar (o se pulsa la tecla [F2]). Tanto el elemento como el conjunto de referencias que posee se subrayan de color verde. El cambio de nombre se realiza automáticamente sobre todas sus ocurrencias para obtener una visualización previa.

Al utilizar este comando, se abre asimismo una ventana que permite indicar el comportamiento que Visual Studio deberá adoptar en esta operación de cambio de nombre. También es esta ventana la que permite efectuar la modificación efectiva del código.



Al hacer clic en el botón **Aplicar**, si se ha marcado la opción **Vista previa de cambios**, se abre una nueva ventana que permite aplicar o anular el cambio de nombre.

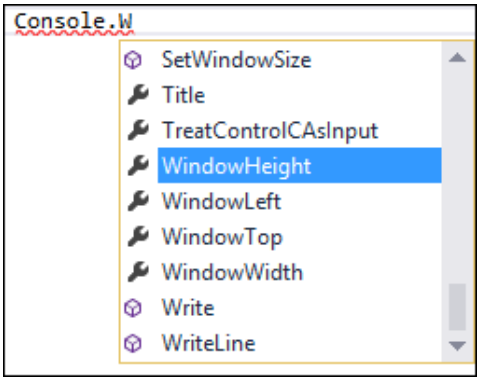


Después de renombrar

Esta operación se realiza pasando el cursor del ratón sobre la bombilla que aparece a la izquierda de la línea con el elemento renombrado **tras una modificación** (o utilizando el atajo de teclado [Alt] + .) y seleccionando la opción **Cambiar el nombre de... a...**.

e. IntelliSense

IntelliSense es el nombre de la funcionalidad de **autocompletar** integrada en el editor de código de Visual Studio. Es contextual, es decir, las sugerencias que se proponen dependen de los elementos de código precedentes. Se utiliza mediante una lista desplegable que aparece en el momento de introducir el texto o mediante la combinación de teclas [Ctrl][Espacio].



Cuando se resalta un elemento de la lista, al pulsar la tecla [Entrar] este se inserta en el lugar donde se encuentre el cursor del ratón.

f. Snippets

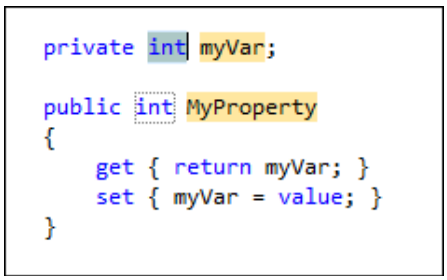
Los code snippets (extractos de código) son porciones de código que pueden incorporarse en el código fuente durante la edición. Permiten escribir muy rápidamente ciertas secciones de código que siempre tienen la misma forma.

Existen tres soluciones para insertar un snippet:

- Mediante la opción **Insertar fragmento de código** del menú contextual del editor.
- Utilizando el atajo de teclado [Ctrl] K, X.
- Seleccionando el nombre de un snippet de la lista que IntelliSense muestra al ir introduciéndolo.

Las dos primeras técnicas muestran una lista de los distintos snippets existentes en Visual Studio. La selección de un elemento de esta lista inserta el código. La selección de un snippet en IntelliSense devuelve, exactamente, el mismo resultado.

En cualquier caso, el editor muestra el código insertado con algunas zonas resaltadas. En el siguiente ejemplo se utiliza el snippet `propfull` para insertar la definición de una variable y de una propiedad asociada.



La modificación del tipo `int` así como el nombre de la variable `myVar` se reflejará, en cascada, en el resto de código del snippet.

Las soluciones

El desarrollo de una aplicación en Visual Studio está organizado según una estructura jerárquica. El elemento de más alto nivel en esta estructura es la solución.

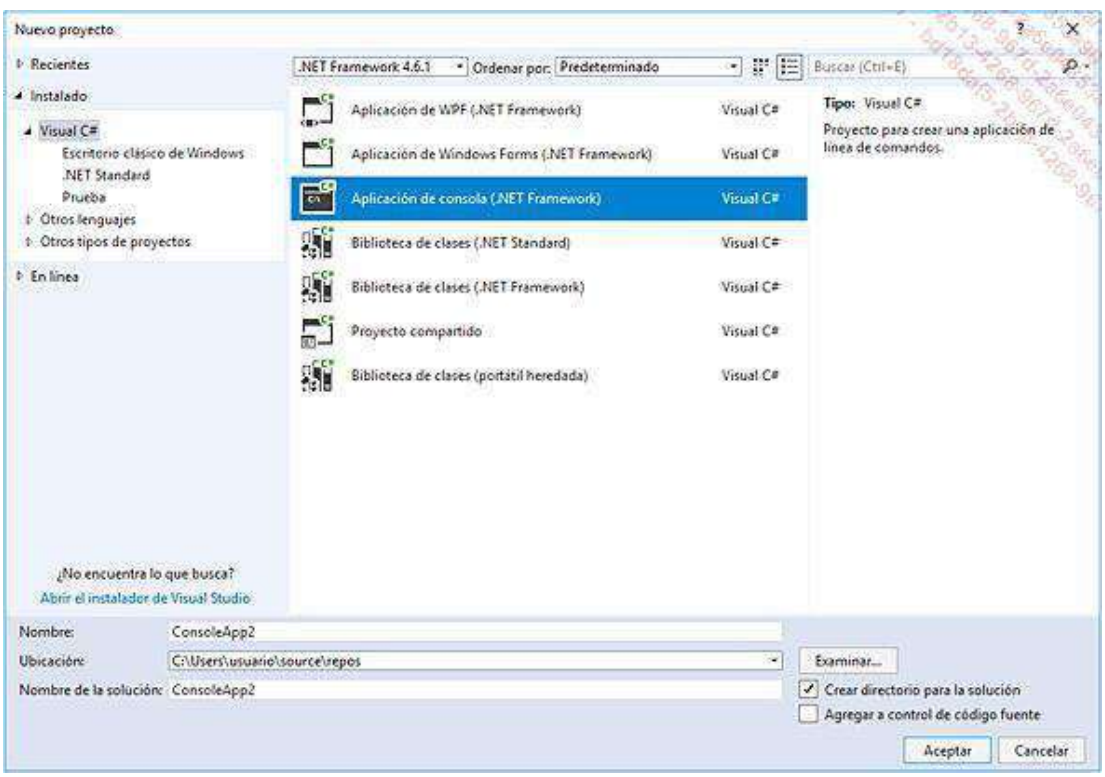
1. Presentación

Una solución es un contenedor lógico que agrupa un conjunto de proyectos distintos que pueden estar vinculados entre sí. Cada uno de estos proyectos contiene uno o varios archivos que permitirán al compilador generar un archivo ejecutable o una biblioteca de clases.

La compilación de una solución puede generar varias aplicaciones y/o archivos .dll formando un conjunto coherente. El código fuente de una aplicación de procesamiento de imágenes, como Paint .NET, podría organizarse en una solución en la que un proyecto permitiera generar el archivo ejecutable y la compilación de los demás proyectos produjese bibliotecas de clases encargadas de proveer los procesamientos utilizados por el ejecutable.

2. Creación de una solución

Cuando crea un nuevo proyecto en Visual Studio, se crea automáticamente una solución que lo contiene. Esta etapa se realiza mediante el menú **Archivo - Nuevo proyecto**. Se abre un cuadro de diálogo.



En esta etapa es posible proveer a Visual Studio una gran cantidad de información:

- La versión del framework .NET que se desea utilizar de entre la lista de versiones instaladas.
- El lenguaje de programación y el modelo de proyecto que se desea utilizar. Ambos elementos están asociados en el árbol de modelos, que es más o menos completo en función de las opciones seleccionadas durante la instalación de Visual Studio.
- El nombre del proyecto.

- La ubicación en la que se desea crear la solución.
- El nombre de la solución. Puede ser diferente del nombre del proyecto.
- La creación de una carpeta para la solución en la ubicación seleccionada.
- La inclusión en el control de código fuente. Aquí se trata de la creación de un repositorio GIT porque en la ventana de opciones (menú **Herramientas - Opciones**), en **Control de código fuente - Selección de complementos**, está seleccionado **Git**. Si la configuración indicara **Visual Studio Team Foundation Server**, la opción indicaría **Agregar al control de código fuente** y necesitaría de la existencia de un repositorio en una solución de gestión de código fuente: **Team Foundation Server** o **Visual Studio Online**.

Una vez validada la información introducida, Visual Studio crea automáticamente todos los archivos iniciales necesarios para el desarrollo en la carpeta indicada y a continuación abre la solución.

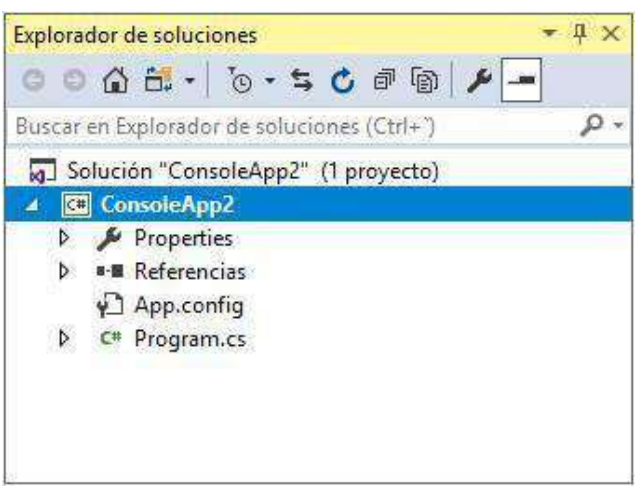
3. Organización

Crear una solución seleccionando el tipo de proyecto **Aplicación de consola** genera los siguientes archivos:

- Un archivo `.sln` que contiene la configuración de la solución con la versión mínima de Visual Studio que puede abrir la solución, la lista de proyectos que contiene, etc. Esta última se actualizará con cada proyecto que se agregue.
- Un archivo `.suo` que contiene la lista de opciones asociadas a un desarrollador para la solución. Este archivo no tiene por qué almacenarse en el sistema de gestión de código fuente.
- Un archivo `.csproj` que representa la configuración del proyecto. Contiene la lista de archivos incluidos en el proyecto.
- Un archivo `App.Config` que contiene ciertos parámetros de configuración para el archivo ensamblado que resulta de la compilación del proyecto.
- Un archivo `Program.cs` que contiene el código fuente de la aplicación.

Una solución puede contener muchos proyectos. Cada uno de estos proyectos se representa mediante un archivo `.csproj` y se referencia en el archivo `.sln` que representa a la solución.

El explorador de soluciones representa la solución en forma de árbol.



4. Acciones disponibles para una solución

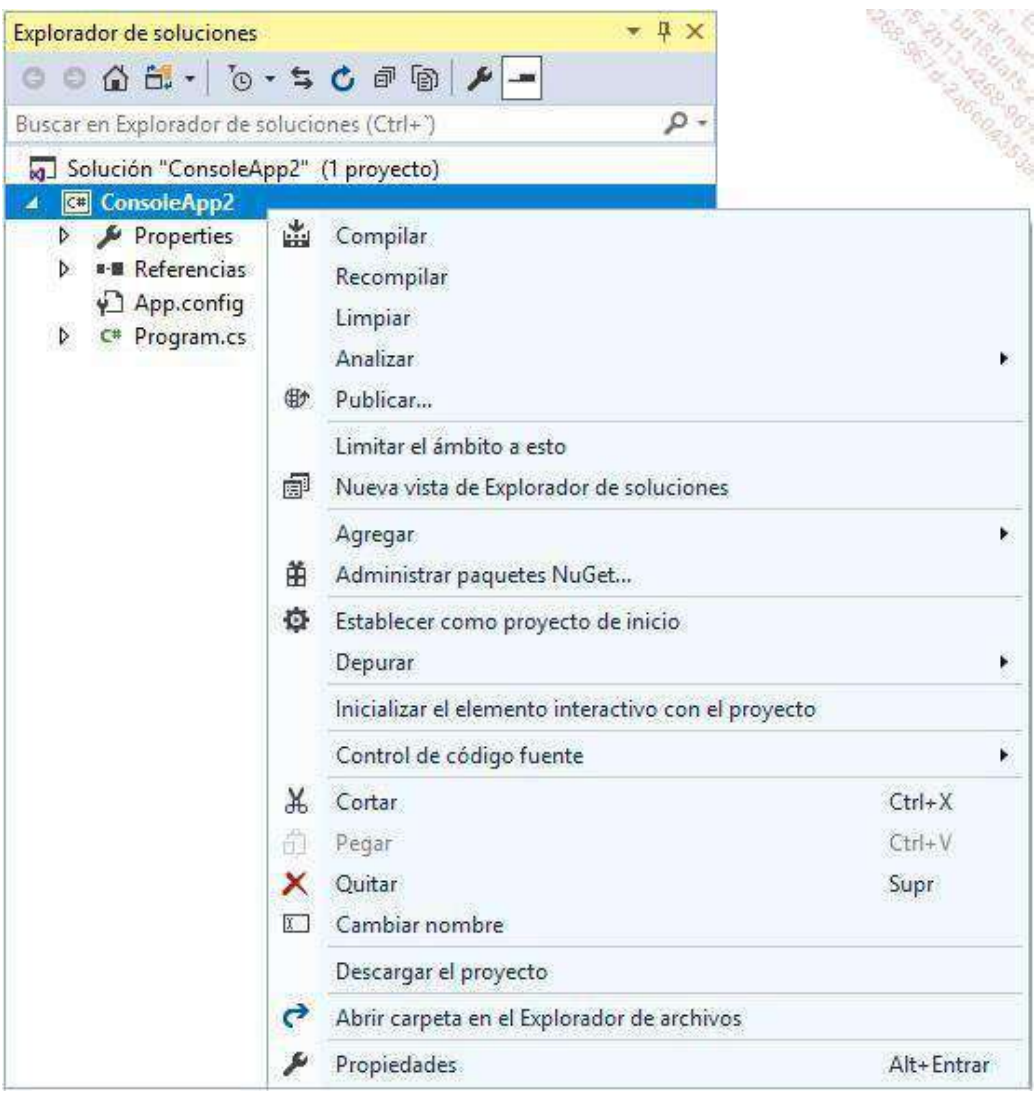
El explorador de soluciones de Visual Studio permite realizar una gran cantidad de acciones sobre las soluciones.

a. Agregar y eliminar un proyecto

La operación que permite agregar un proyecto a una solución se realiza con ayuda del menú contextual, que se abre haciendo clic con el botón derecho sobre el nodo raíz del explorador de soluciones. Conviene seleccionar la opción de menú **Agregar - Nuevo proyecto** y, a continuación, proveer los parámetros necesarios para crear el proyecto en la ventana emergente abierta. Esta acción está disponible, también, mediante el menú **Archivo - Agregar - Nuevo proyecto**.

También es posible agregar un proyecto existente a una solución. Para ello se realiza la misma operación que para la creación de un proyecto, pero seleccionando la opción **Proyecto existente**. Esta operación no copia el proyecto original. Cualquier modificación de los archivos afectará a todas las soluciones que utilicen el proyecto.

Los proyectos se pueden eliminar mediante la opción **Eliminar** del menú contextual, que se abre haciendo clic con el botón derecho sobre la raíz del proyecto (y no de la solución). Esta operación elimina únicamente el proyecto de la solución: el proyecto sigue almacenado en el disco duro y puede agregarse de nuevo a la solución.



b. Creación de una carpeta de soluciones

Las carpetas de soluciones son elementos lógicos de estructuración. Permiten agrupar proyectos agregando uno o

varios niveles jerárquicos. Pueden, también, contener archivos que no pertenezcan a un proyecto pero que sea conveniente agregar a la solución.

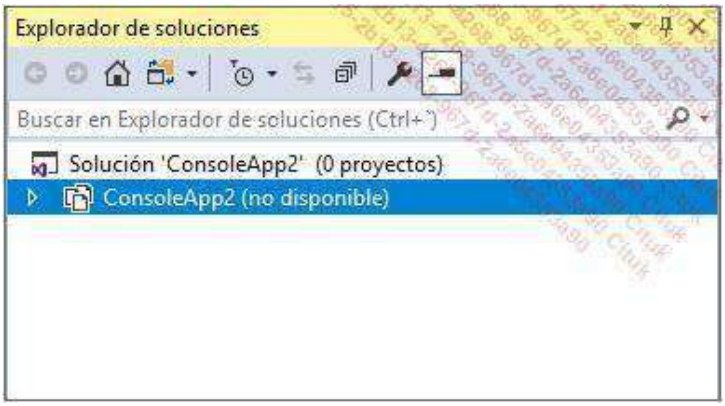
El menú contextual de la solución presenta una opción que permite agregar una carpeta de soluciones. Se encuentra en **Agregar - Nueva carpeta de soluciones**.

La creación de una carpeta de soluciones no se refleja en el sistema de archivos: no se crea, en efecto, ninguna carpeta suplementaria en la carpeta de la solución.

Del mismo modo que es posible agregar un proyecto a la solución, también es posible agregar un proyecto en una carpeta de soluciones y arrastrar y soltar un proyecto de la solución sobre una carpeta de soluciones.

c. Carga y descarga de un proyecto

En ocasiones puede resultar interesante excluir temporalmente un proyecto de una solución. Para ello, se utiliza la opción **Descargar el proyecto** en el menú contextual del proyecto correspondiente. Una vez realizada esta acción, el explorador de soluciones marca el proyecto como no disponible e impide la modificación de los archivos que contiene.

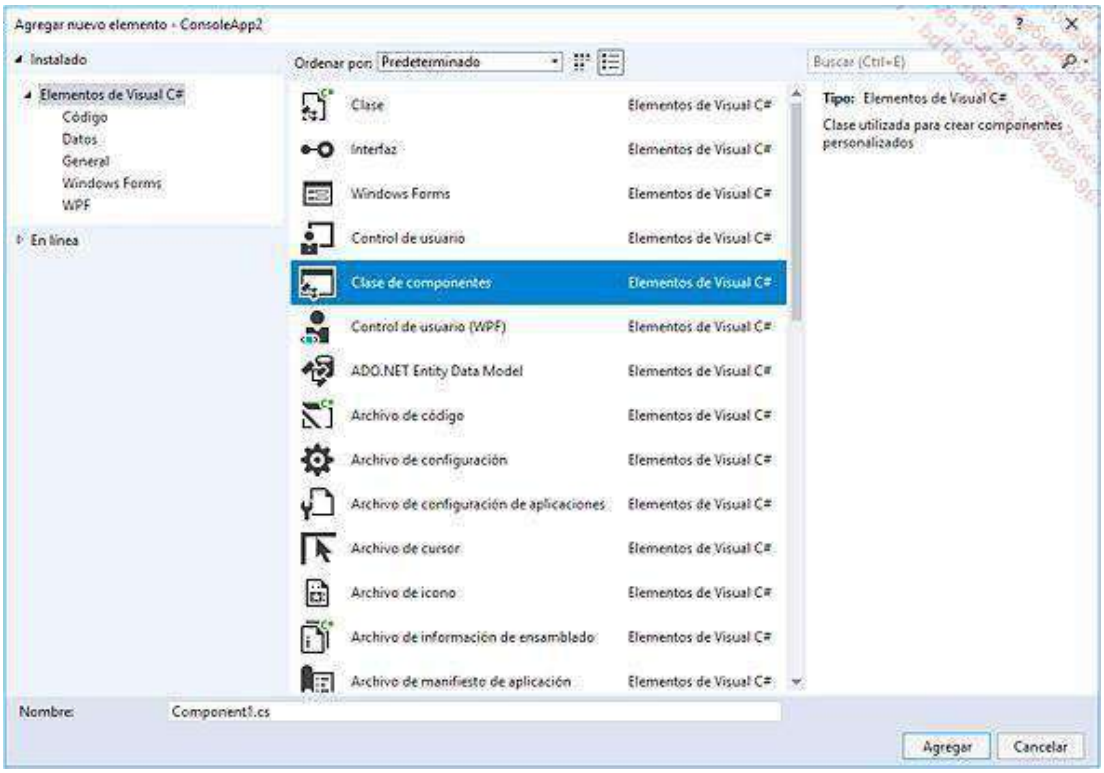


Cada proyecto descargado tiene un menú contextual restringido que permite volver a cargarlo en el proyecto.

d. Creación de un archivo

El menú contextual de la solución ofrece la posibilidad de agregar un archivo mediante el sub-menú **Agregar - Nuevo elemento** y **Agregar - Elemento existente**. Ambas opciones pueden ejecutarse también con los atajos de teclado [Ctrl][Mayús] A y [Alt][Mayús] A.

Al agregar un elemento se abre un cuadro de diálogo que permite seleccionar el tipo de archivo que se desea agregar, así como su nombre.



Si se escoge abrir un archivo existente se abre una ventana de selección de archivos clásica.

e. Compilar la solución

El uso de las opciones **Compilar solución** y **Recompilar solución** realizan una compilación de todos los archivos de la solución. La diferencia entre ambas acciones reside en la gestión de los proyectos no modificados desde la última compilación.

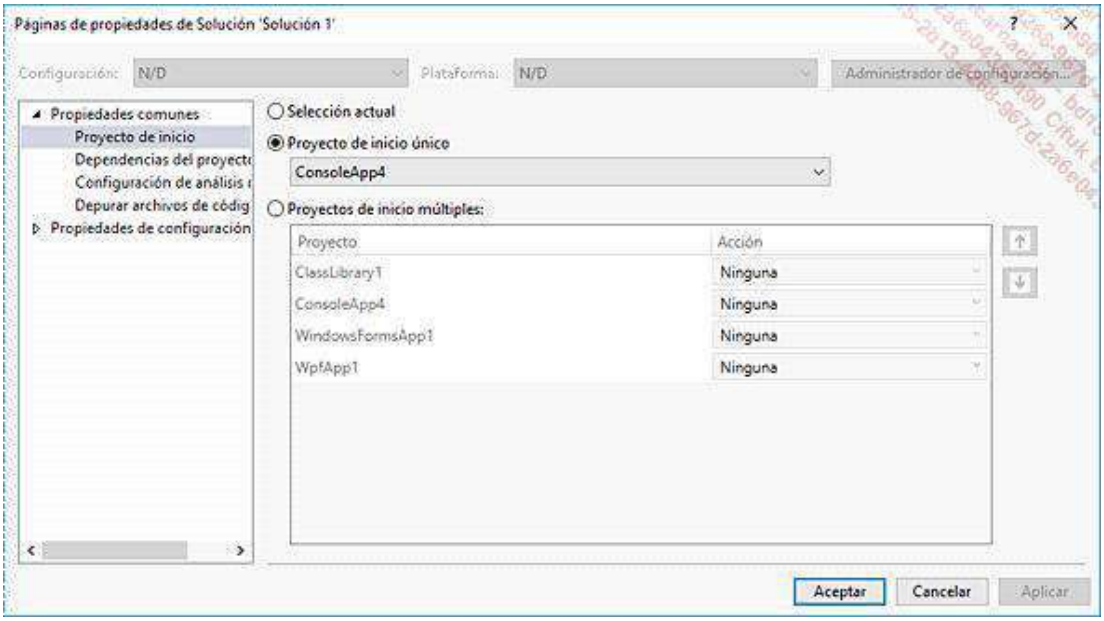
Cuando un proyecto no ha sido modificado, la primera opción no realiza una recompilación. La segunda opción, cuando se selecciona, fuerza la recompilación de todos los proyectos, hayan sido modificados o no.

5. Configuración de la solución

Las soluciones disponen de propiedades que permiten influir en su comportamiento durante la generación o la ejecución. Estos parámetros pueden modificarse mediante el cuadro de diálogo del sub-menú **Propiedades** que se encuentra en el menú contextual de la solución.

a. Establecer proyectos de inicio

Esta página de propiedades permite definir qué proyectos deben ejecutarse al iniciar la solución.



Existen tres opciones disponibles para este parámetro.

Selección actual

Esta opción indica que el proyecto en curso se ejecutará tras el inicio de la solución.

Proyecto de inicio único

Se muestra una lista desplegable que contiene el nombre de cada proyecto de la solución. El proyecto seleccionado en la lista se ejecutará al iniciar la solución.

Cuando se selecciona la opción, el nombre del proyecto marcado como proyecto de inicio se muestra en negrita en el explorador de soluciones. Es posible, también, modificar el proyecto de inicio único directamente en el explorador de soluciones abriendo el menú contextual de un proyecto y seleccionando la opción **Establecer como proyecto de inicio**.

Proyectos de inicio múltiples

Una tabla muestra la lista de proyectos de la solución. Es posible especificar la acción que se desea llevar a cabo al iniciar cada uno de ellos. Existen tres opciones posibles para cada proyecto:

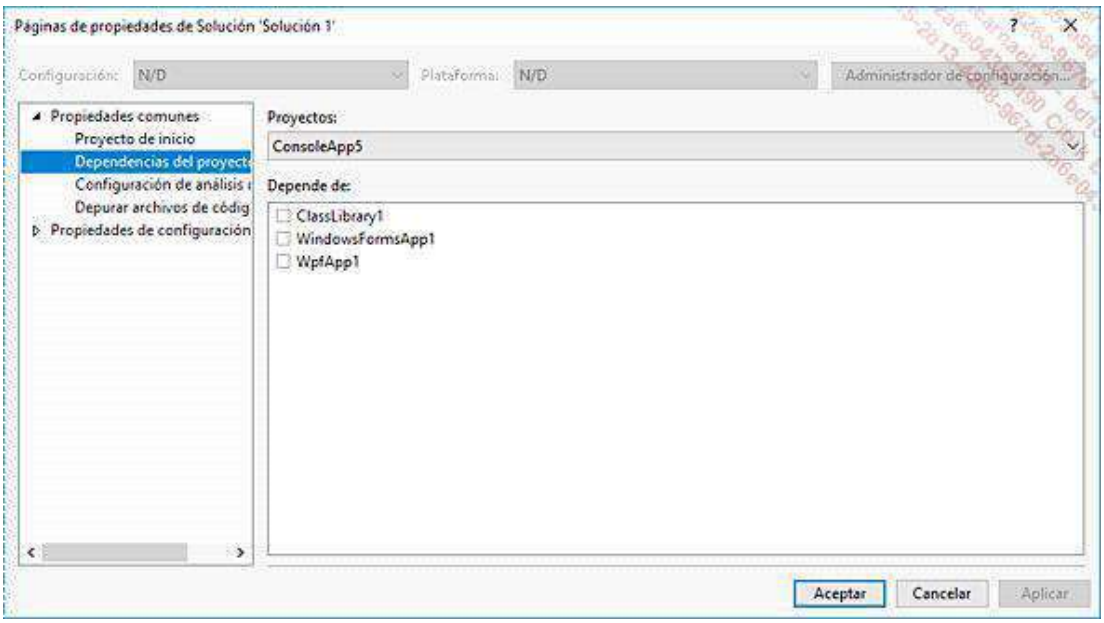
- **Ninguna**
- **Iniciar**
- **Iniciar sin depurar**

Para modificar el orden de inicio de cada proyecto seleccionado, basta con modificar el orden de los elementos en la tabla. Para ello, se muestran dos botones que permiten subir y bajar el orden del proyecto en la lista, respectivamente.

b. Dependencias del proyecto

La compilación de un proyecto puede requerir que algún otro proyecto se haya compilado previamente. La pantalla

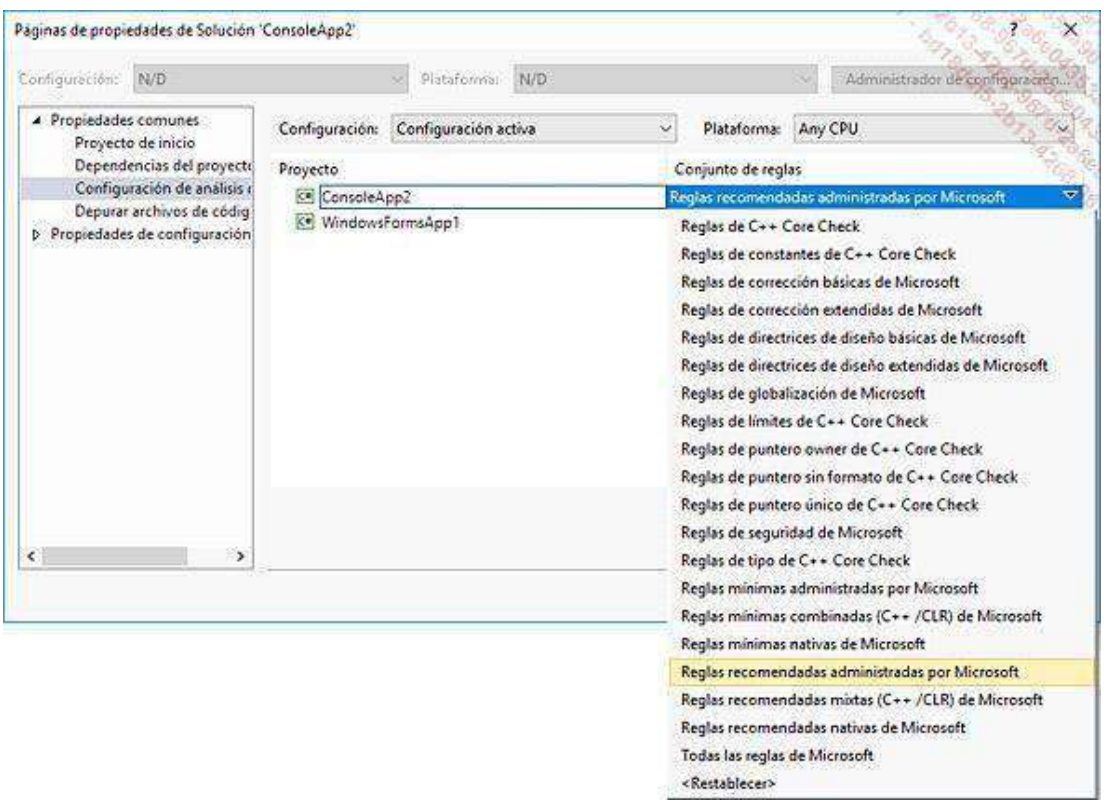
de gestión de dependencias permite configurar estas dependencias.



Para configurar las dependencias de un proyecto es preciso seleccionarlo en la lista desplegable. A continuación, se muestra la lista de otros proyectos y se ofrece la posibilidad de marcar cada uno de ellos. Las compilaciones de proyectos generarán previamente cada una de las dependencias indicadas por las opciones marcadas.

c. Configuración de análisis de código

Esta página permite configurar las reglas que se utilizan para analizar el código de cada uno de los proyectos de la solución.

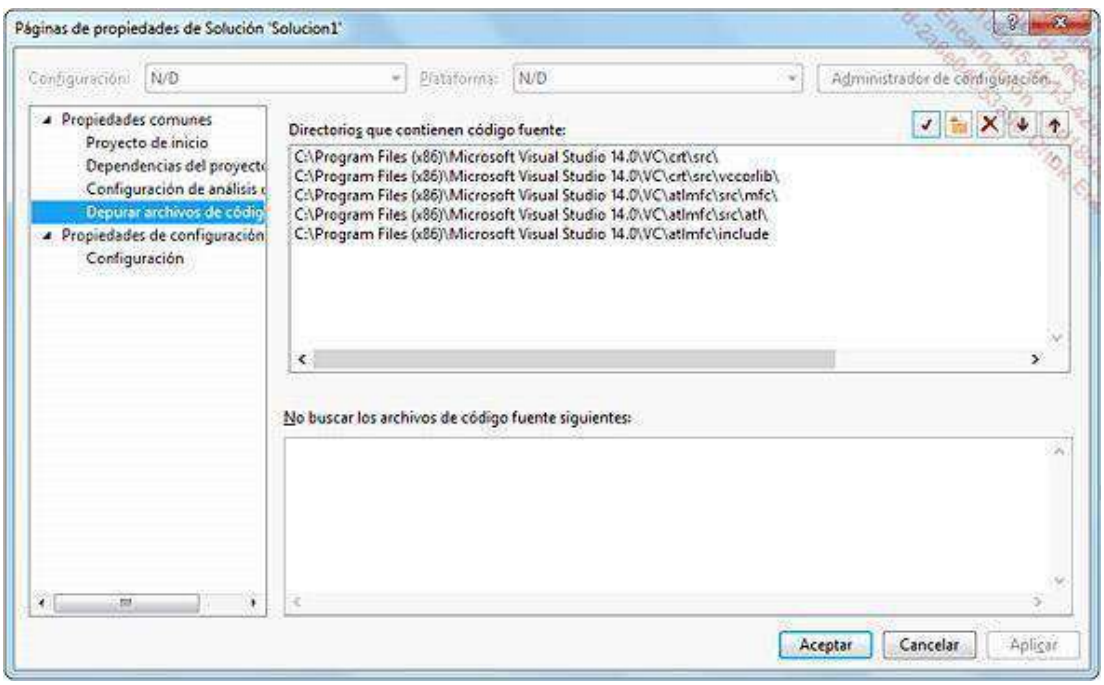


Para cada uno de los proyectos existe una lista desplegable que permite indicar la regla que se desea aplicar. La ventana **Lista de errores** de Visual Studio muestra los errores o puntos a monitorizar. Es posible consultar el detalle de cada regla en el sitio MSDN.

En el momento de escribir estas líneas, las reglas se describen en la página **Referencia del conjunto de reglas Análisis de código** que se encuentra en la dirección: http://msdn.microsoft.com/es-es/library/dd264925.aspx

d. Depurar archivos de código fuente

El depurador de Visual Studio necesita acceder al archivo de código fuente que se está depurando. La página de propiedades **Depurar archivos de código fuente** permite indicar al entorno cuáles son las carpetas que se desea inspeccionar para buscar estos archivos de código fuente.

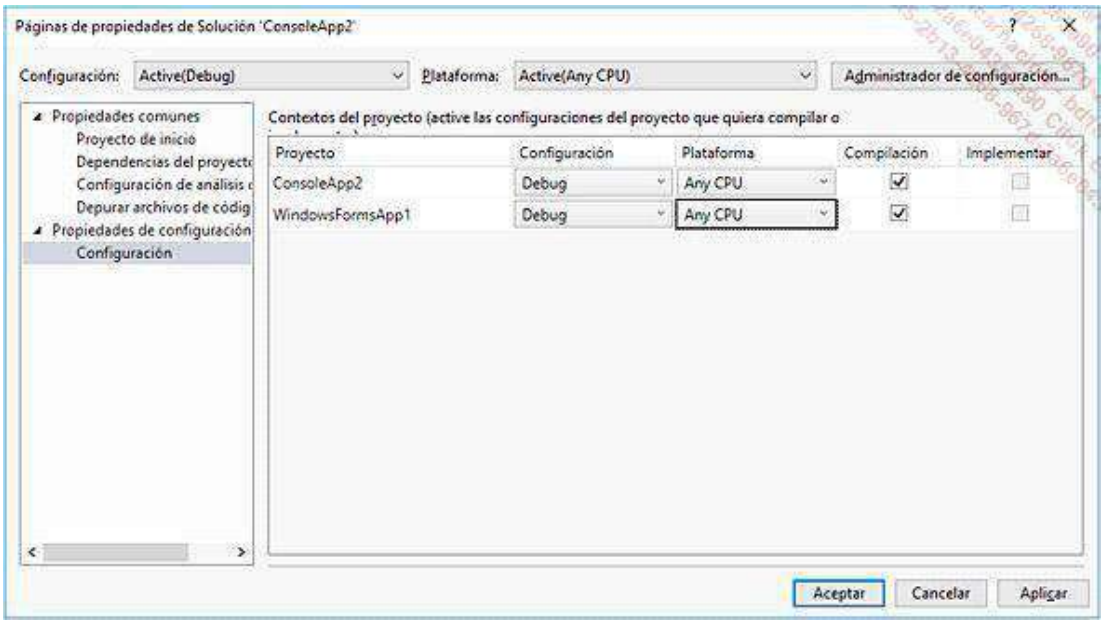


La lista de **Directorios que contienen código fuente** muestra la lista de carpetas en las que se buscará el código fuente. Esta lista puede gestionarse mediante la barra de herramientas situada sobre la lista.

La lista **No buscar los archivos de código fuente siguientes** permite excluir ciertos archivos de la búsqueda.

e. Configuración

Esta última página de propiedades permite definir la forma en que se generarán los proyectos de la solución. Por defecto, existen dos configuraciones en Visual Studio: **Debug** y **Release**. La configuración puede especificarse para la solución completa o para cada uno de los proyectos individualmente.



La configuración **Debug** se utiliza generalmente en la fase de desarrollo y pruebas, mientras que la configuración **Release** se utiliza en la generación final de un proyecto.

También es posible definir la arquitectura de hardware de destino en cada uno de los proyectos. Por defecto, está seleccionado el valor **Any CPU**, aunque es posible seleccionar los valores **ARM**, **x86** o **x64**.

Los proyectos

Un proyecto reúne el código fuente de una aplicación o una biblioteca de clases. Puede contener archivos de código, archivos de definición de interfaz gráfica o recursos, gráficos o no.

1. Creación de un proyecto

En Visual Studio, la creación de un proyecto se realiza mediante el cuadro de diálogo **Nuevo proyecto**, que podemos abrir desde el menú contextual de una solución (**Agregar - Nuevo proyecto**) o mediante el menú de Visual Studio (**Archivo - Agregar - Nuevo proyecto**). En ambos casos, la creación de un proyecto requiere seleccionar una plantilla a partir de la cual se genera la estructura.

Visual Studio ofrece por defecto numerosas plantillas de proyecto. Cada una de estas plantillas contiene los elementos básicos para la creación del proyecto del tipo correspondiente, así como las referencias más útiles para ese tipo de proyecto.

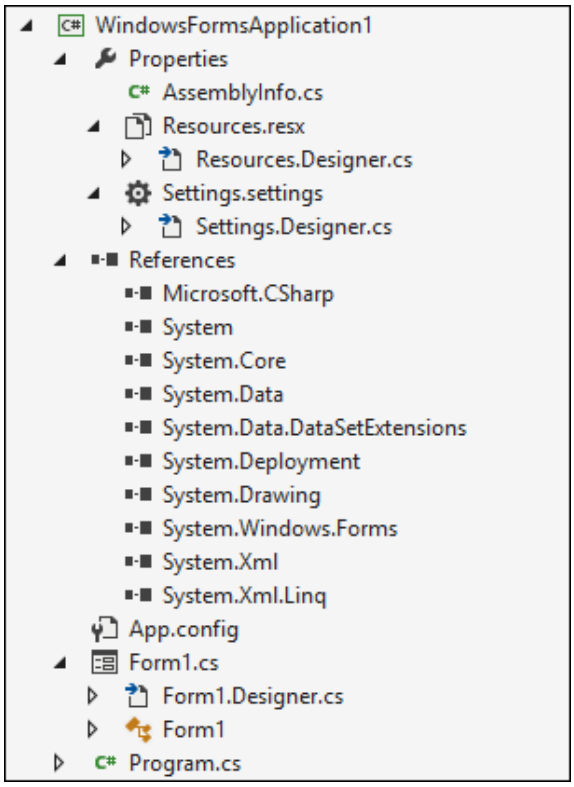
El cuadro de diálogo **Nuevo proyecto**, cuando se selecciona **Visual C# - Escritorio clásico Windows**, contiene una lista de once plantillas. Detallemos estas plantillas para comprender mejor su utilidad y sus diferencias.

Aplicación Windows Forms

Este tipo de proyecto permite construir aplicaciones de ventanas tradicionales de Windows utilizando la tecnología **Windows Forms**.

La plantilla del proyecto agrega nuevos archivos al proyecto tras su creación:

- `AssemblyInfo.cs`: este archivo de código contiene las propiedades del ensamblado que resultará de la compilación del proyecto (número de versión, nombre del editor, etc.).
- `Resources.resx` y `Resources.Designer.cs`: el primero permite editar los recursos de la aplicación, mientras que el segundo lo genera Visual Studio para encapsular el acceso a los recursos.
- `Settings.settings` y `Settings.Designer.cs`: estos archivos se utilizan de la misma manera que los archivos de recursos, pero están destinados a almacenar parámetros relativos al usuario o a la aplicación en su conjunto.
- `App.config`: se trata de un archivo con formato XML que se asocia al ensamblado generado por el proyecto. Contiene los parámetros de configuración de la aplicación.
- `Form1.cs` y `Form1.Designer.cs`: estos archivos definen una ventana de aplicación editable mediante un diseñador visual de Visual Studio. El archivo `Form1.Designer.cs` contiene el código generado por el entorno de desarrollo tras la edición del formulario, mientras que `Form1.cs` contiene el código escrito por el desarrollador. Este último archivo contendrá, entre otros, los controladores de eventos relativos a la ventana.
- `Program.cs`: contiene el punto de entrada de la aplicación.



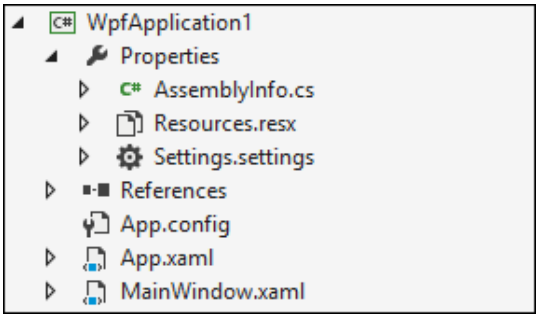
El árbol del proyecto presenta también un elemento llamado **Referencias**. Se trata de una carpeta lógica, presente en todos los proyectos .NET, que enumera los ensamblados que se utilizarán en el proyecto. Esta lista puede extenderse con ensamblados procedentes del framework .NET o de fuentes de terceros en función de las necesidades.

Aplicación WPF

El tipo de proyecto **Aplicación WPF** proporciona las bases para la creación de una aplicación de ventanas que utiliza la tecnología **Windows Presentation Foundation**. Como la plantilla de proyecto anterior, la plantilla de aplicación WPF crea un proyecto que contiene nueve archivos.

Encontramos los archivos de descripción del ensamblado, de gestión de recursos y parámetros, así como el archivo de configuración de la aplicación, todos ellos con el mismo rol que en una aplicación Windows Forms.

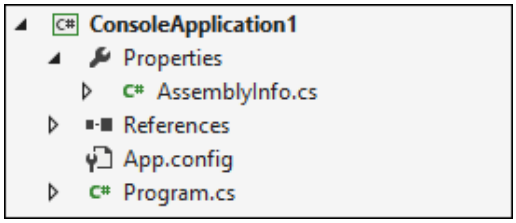
El archivo `Program.cs` no existe en este modelo, ha sido reemplazado por el archivo `App.xaml`. La ventana inicial, llamada por defecto `MainWindow.xaml`, contiene la descripción de la interfaz gráfica mediante lenguaje XAML, mientras que el archivo `MainWindow.xaml.cs` es su archivo de code-behind asociado. Este último contendrá los controladores de eventos relativos a la ventana.



Aplicación de consola

Este tipo de proyecto permite crear una aplicación que utiliza la línea de comandos. No contiene, por tanto, ninguna interfaz gráfica. Esto supone su punto fuerte pero también su debilidad, una aplicación de consola es ideal para realizar pruebas con Visual C# sin tener que asociar una interfaz de usuario, aunque actualmente las interfaces de usuario deben ser agradables y prácticas. Esta austeridad gráfica así como la dificultad de uso vinculada a la entrada/salida por teclado hacen que este tipo de proyectos estén obsoletos.

Un proyecto de este tipo contiene inicialmente tres archivos: `AssemblyInfo.cs`, `App.config` y `Program.cs`.



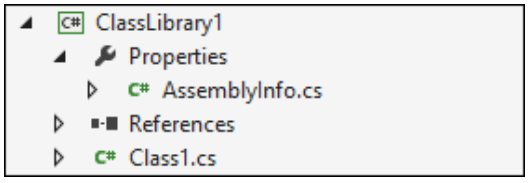
Proyecto compartido

Un proyecto compartido ofrece la posibilidad de integrar una misma sección de código en varios proyectos. Todos los proyectos que poseen una referencia a un proyecto compartido pueden acceder al código que este contiene y el resultado de su compilación integra, en su interior, el código en cuestión. Los proyectos compartidos son, por lo tanto, la manera perfecta para factorizar el código de una solución y mejoran el mantenimiento de las aplicaciones, evitando la duplicación.

Al crearlos, los proyectos de este tipo están vacíos.

Biblioteca de clases

La plantilla de proyecto **Biblioteca de clases** tiene como objetivo crear archivos .dll que puedan utilizarse desde ejecutables .NET. Un proyecto de este tipo es extremadamente sencillo: contiene inicialmente un archivo `AssemblyInfo.cs` y un archivo de código llamado, por defecto, `Class1.cs`.



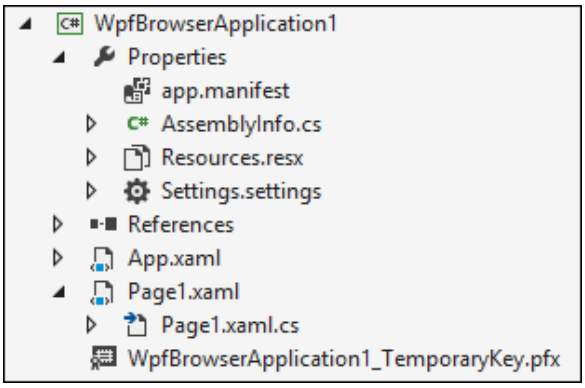
Aplicación de explorador WPF

Se puede acceder a las aplicaciones WPF a través de un navegador web siempre que se utilice una configuración específica. La plantilla de proyecto **Aplicación de explorador WPF** crea la configuración necesaria así como una estructura de archivos similar, si no idéntica, a la de una aplicación WPF clásica.

El proyecto de aplicación de navegador WPF incluye un archivo llamado `app.manifest` que contiene ciertos parámetros necesarios para la ejecución de la aplicación.

Los archivos `MainWindow.xaml` y `MainWindow.xaml.cs` se remplazan, aquí, por un archivo llamado `Page1.xaml` y por el archivo de code-behind asociado.

Por último, este proyecto contiene un archivo cuyo nombre termina por `_TemporaryKey.pfx`. Este archivo se encarga de la firma digital de la aplicación (en curso de desarrollo) antes de su despliegue.



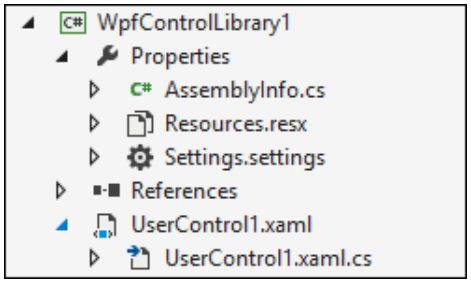
Biblioteca de controles de usuario de WPF

Un control de usuario representa una porción de interfaz gráfica cuyo aspecto y comportamiento asociado se externalizan en un elemento reutilizable. Este elemento está compuesto por uno o varios controles definidos en un archivo `.xaml`. El comportamiento de estos componentes se codifica en un archivo de code-behind asociado al control de usuario.

Las bibliotecas de controles de usuario contienen uno o varios controles de este tipo y permiten, de este modo, extender el cuadro de herramientas de Visual Studio.

Como ocurre con todos los tipos de proyecto que hemos visto hasta ahora, un proyecto **Biblioteca de controles de usuario** contiene un archivo `Assembly.cs` con información relativa a la descripción del ensamblado que se generará para el proyecto. Contiene también los archivos destinados a almacenar recursos y parámetros.

Por defecto, se crea un control de usuario vacío. Los archivos asociados son `UserControl1.xaml` y `UserControl1.xaml.cs`.

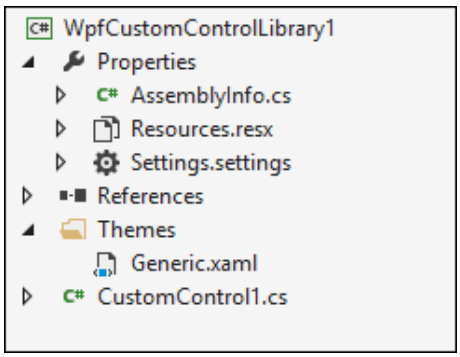


Biblioteca de controles personalizados de WPF

Como con la plantilla de proyecto anterior, la plantilla **Biblioteca de controles personalizados de WPF** permite extender el número de componentes disponibles para el desarrollo creando nuevas piezas y controles. Es posible también extender un control existente agregándole funcionalidades. En cualquier caso, es necesario redefinir el aspecto visual del control.

Un proyecto de **Biblioteca de controles personalizados de WPF** incluye, además de los archivos de recursos, parámetros y descripción del ensamblado, la definición de un primer control personalizado en un archivo llamado `CustomControl1.cs`. El código XAML que define el aspecto visual del control se encuentra en un archivo llamado `Generic.xaml`, dentro de una sub-carpeta llamada `Themes`. El nombre de estos dos elementos debe,

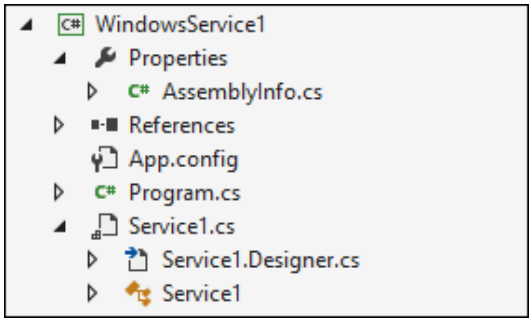
obligatoriamente, ser el indicado.



Servicio de Windows

Esta plantilla de proyecto permite desarrollar aplicaciones que ejecutará directamente el sistema operativo como tarea de fondo. Este tipo de programa no posee una interfaz de usuario, lo que implica que la interacción con el usuario debe implementarse de manera separada y es preciso prever una capa de comunicación para transmitir y recibir información.

El archivo `Service1.cs` contiene el esqueleto de un servicio de Windows y el archivo `Service1.Designer.cs` contiene puntos de código fuente autogenerados automáticamente por Visual Studio.



Biblioteca de controles de Windows Forms

La plantilla **Biblioteca de controles de Windows Forms** es el equivalente a la plantilla **Biblioteca de controles de usuario de WPF** para la tecnología Windows Forms. También permite crear nuevos controles combinando varios componentes existentes.

Por defecto, un proyecto de este tipo contiene un archivo de descripción de ensamblado y un esqueleto de control de usuario. La definición visual de este control se genera mediante el diseñador visual de Visual Studio en el archivo `UserControl1.Designer.cs` mientras que el archivo `Usercontrol1.cs` contiene la lógica del control.

Proyecto vacío

Este tipo de proyecto debería utilizarse cuando desee crear su propio tipo de proyecto. Solo se crean dos archivos: el archivo `.csproj`, que define su naturaleza y su contenido, y un archivo de configuración por defecto llamado `App.config`.

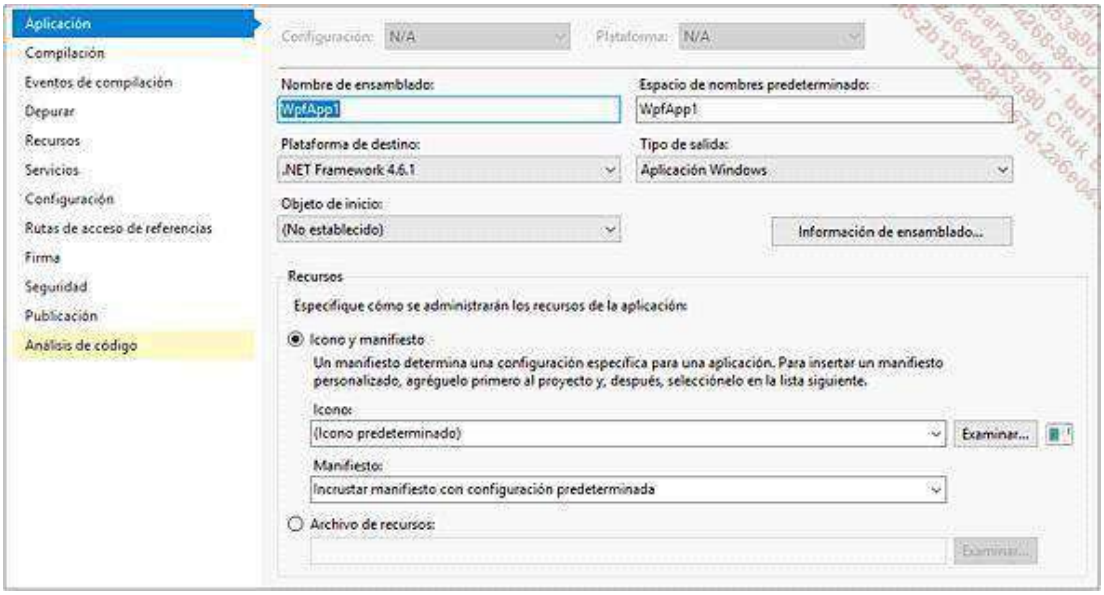
2. Propiedades de un proyecto

Los proyectos son elementos fundamentales en el desarrollo de aplicaciones con Visual Studio. A este respecto, poseen un gran número de propiedades que permiten modificar su comportamiento durante la fase de diseño, compilación o ejecución de las aplicaciones. Estas propiedades están accesibles mediante su nombre en el menú contextual de los proyectos. Existen varias pestañas disponibles en la ventana de propiedades.

El presente manual trata los fundamentos del uso de Visual Studio y C#, algunas pestañas no se describen debido a que no se corresponden con esta fase de su aprendizaje.

a. Aplicación

Esta pestaña presenta las propiedades relativas al comportamiento general de la aplicación.



Nombre de ensamblado

El valor de esta propiedad determina el nombre del archivo resultante de la compilación del proyecto. Por defecto, este nombre coincide con el del proyecto, pero puede modificarse sin afectar al nombre del proyecto.

Plataforma de destino

La selección de un valor en esta lista modifica el framework .NET de destino para el proyecto. Las funcionalidades disponibles para el desarrollador dependen de esta propiedad. Por defecto, su valor se corresponde con el que se haya seleccionado durante la creación del proyecto.

Espacio de nombres predeterminado

Esta propiedad define el espacio de nombres raíz para los elementos del proyecto. A cada archivo de código que se agrega al proyecto se le asignará automáticamente un espacio de nombres correspondiente a su ubicación en la jerarquía del proyecto. Por defecto, el valor de esta propiedad se corresponde con el nombre del proyecto.

Tipo de salida

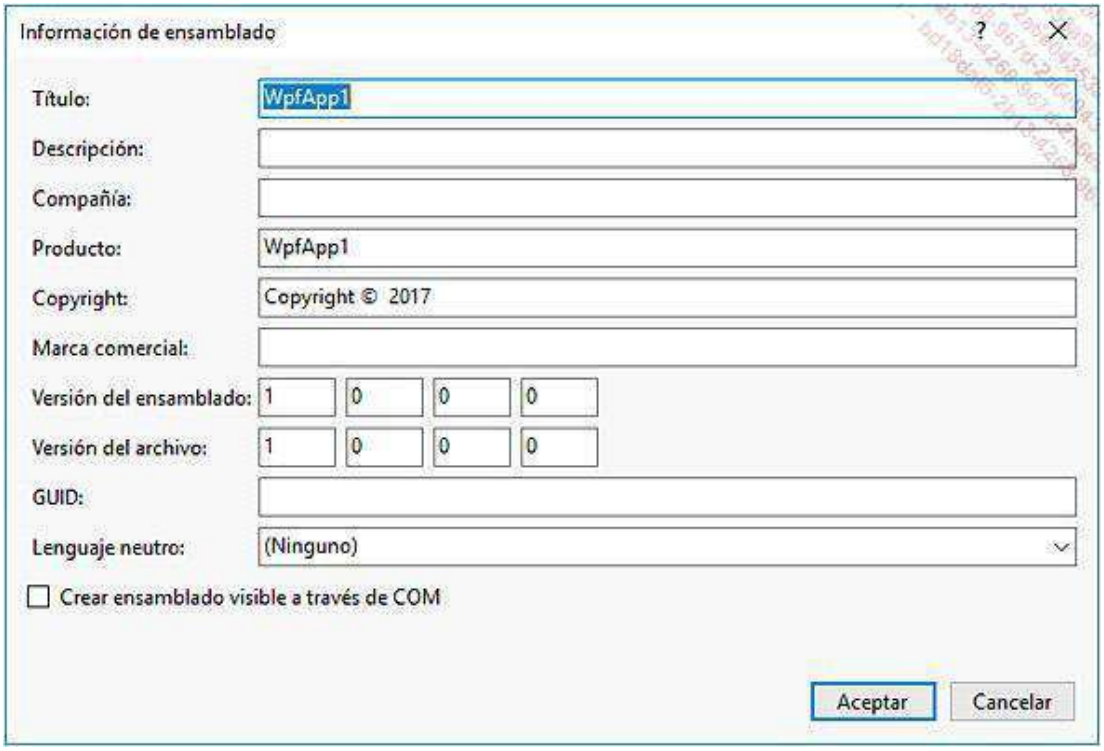
El tipo de resultado de una aplicación se corresponde con el tipo de archivo que se genera tras su compilación. Este valor está vinculado, generalmente, con la plantilla de proyecto utilizada y habitualmente no se modifica, puesto que el tipo de resultado depende en gran medida del código fuente asociado al proyecto.

Objeto de inicio

El objeto de inicio define el tipo que contiene el punto de entrada a la aplicación. Esta propiedad está disponible únicamente para aquellos tipos de proyecto que puedan ejecutarse, como por ejemplo una aplicación de Windows Forms. En el caso de una biblioteca de clases, esta propiedad contiene el valor (**Sin definir**).

Información de ensamblado

El botón **Información de ensamblado** abre un cuadro de diálogo que permite editar el contenido del archivo `AssemblyInfo.cs` de una manera más gráfica.



Recursos

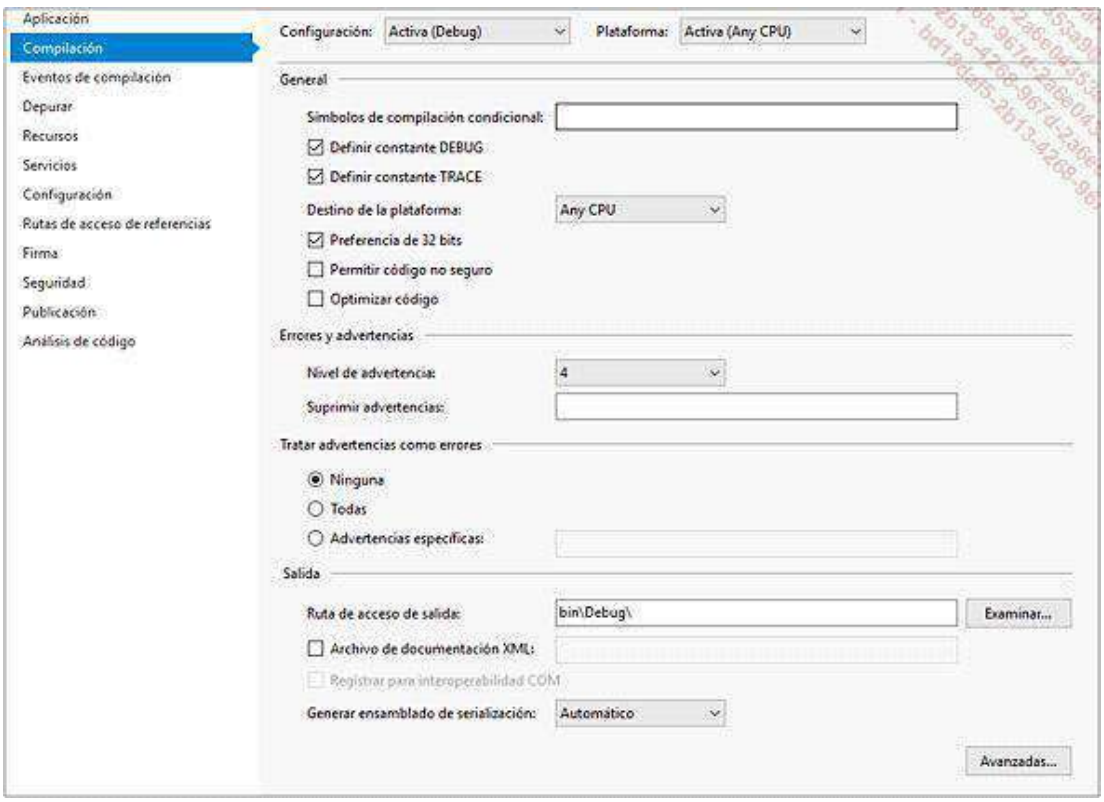
Esta sección permite seleccionar el modo de gestión de ciertos recursos de la aplicación. Se ofrecen dos posibilidades al desarrollador.

La opción **Icono y manifiesto** permite modificar el icono de la aplicación para personalizar su representación en la barra de tareas o en el explorador de Windows. Es posible, también, optar por incorporar un manifiesto en la aplicación. Los manifiestos se utilizan para determinar el nivel de ejecución requerido por la aplicación. Es posible incorporar automáticamente un manifiesto que requiere permisos del usuario que ejecuta la aplicación.

La opción **Archivo de recursos** permite integrar un archivo de recursos Win32 estándar en la aplicación. Se utilizará el icono de aplicación almacenado en este archivo.

b. Compilación

La página de propiedades **Compilación** presenta varias opciones que permiten interactuar sobre la compilación del proyecto.



Las dos listas desplegables **Configuración** y **Plataforma** permiten definir el par configuración/plataforma sobre la que se editarán las propiedades.

Símbolos de compilación condicional

Esta propiedad permite especificar una o varias constantes que pueden evaluarse mediante directivas del preprocesador `#if`. Ciertas secciones de código pueden, de este modo, compilarse únicamente si se define un símbolo de compilación.

Cuando se definen varias constantes es preciso separarlas mediante un espacio.

Definir constante DEBUG

Cuando se marca esta opción, se define automáticamente una constante de compilación llamada `DEBUG`.

Definir constante TRACE

Cuando se marca esta opción, se define automáticamente una constante de compilación llamada `TRACE`.

Destino de la plataforma

Esta lista desplegable permite especificar la arquitectura de procesador de destino para la aplicación. El valor por defecto de esta propiedad es **Any CPU**, que permite garantizar la portabilidad del código sobre cualquier tipo de arquitectura soportada. Es posible, no obstante, seleccionar **x86** o **x64** si conocemos previamente el entorno de

destino de la aplicación.

La opción **Preferencia de 32 bits** está disponible únicamente si se selecciona la plataforma de destino **Any CPU** y permite indicar que, independientemente de la máquina sobre la que se ejecute el código, la aplicación debe funcionar como si la arquitectura de la máquina fuera de 32 bits.

Permitir código no seguro

El código no seguro (*code unsafe*, en inglés) utiliza el concepto avanzado de los punteros. El framework .NET y C# permiten el uso de punteros, pero estos elementos pueden causar problemas, por lo que es necesario que el desarrollador acepte explícitamente sus riesgos.

Optimizar código

Esta opción habilita o deshabilita la optimización que realiza el compilador para hacer que el código sea más eficaz.

Nivel de advertencia

El nivel de advertencia define el tipo de mensaje que debe mostrarse en el compilador cuando se reconoce un problema. El nivel 0 se corresponde con la eliminación completa de estos mensajes de advertencia, mientras que el nivel 4 es el más verboso: se muestran todos los mensajes de advertencia, incluso aquellos menos críticos.

Suprimir advertencias

Este parámetro permite suprimir todas las advertencias de ciertos tipos. Para ello, conviene indicar los códigos numéricos asociados a cada advertencia que no se desea visualizar, separados por comas o por puntos y coma.

Tratar advertencias como errores

Es posible considerar todas o parte de las advertencias devueltas por el compilador como errores bloqueantes. Para ello, es preciso seleccionar la opción apropiada:

- **Ninguna**: las advertencias no se consideran jamás como errores.
- **Todas**: el compilador no tolera ninguna advertencia.
- **Advertencias específicas**: los códigos numéricos de las advertencias indicadas se consideran como errores.

Ruta de accesso de salida

Este parámetro indica la carpeta donde se almacenarán los archivos generados por el compilador. Por defecto, la carpeta es **\bin\Debug** o **\bin\Release** en función de la configuración seleccionada.

Archivo de documentación XML

El compilador C# es capaz de generar un archivo de documentación a partir de comentarios con un formato determinado presentes en el código fuente. Este archivo puede, a continuación, consumirse en herramientas como **Sandcastle** para generar una documentación en formato HTML (u otro).

Registrar para interoperabilidad COM

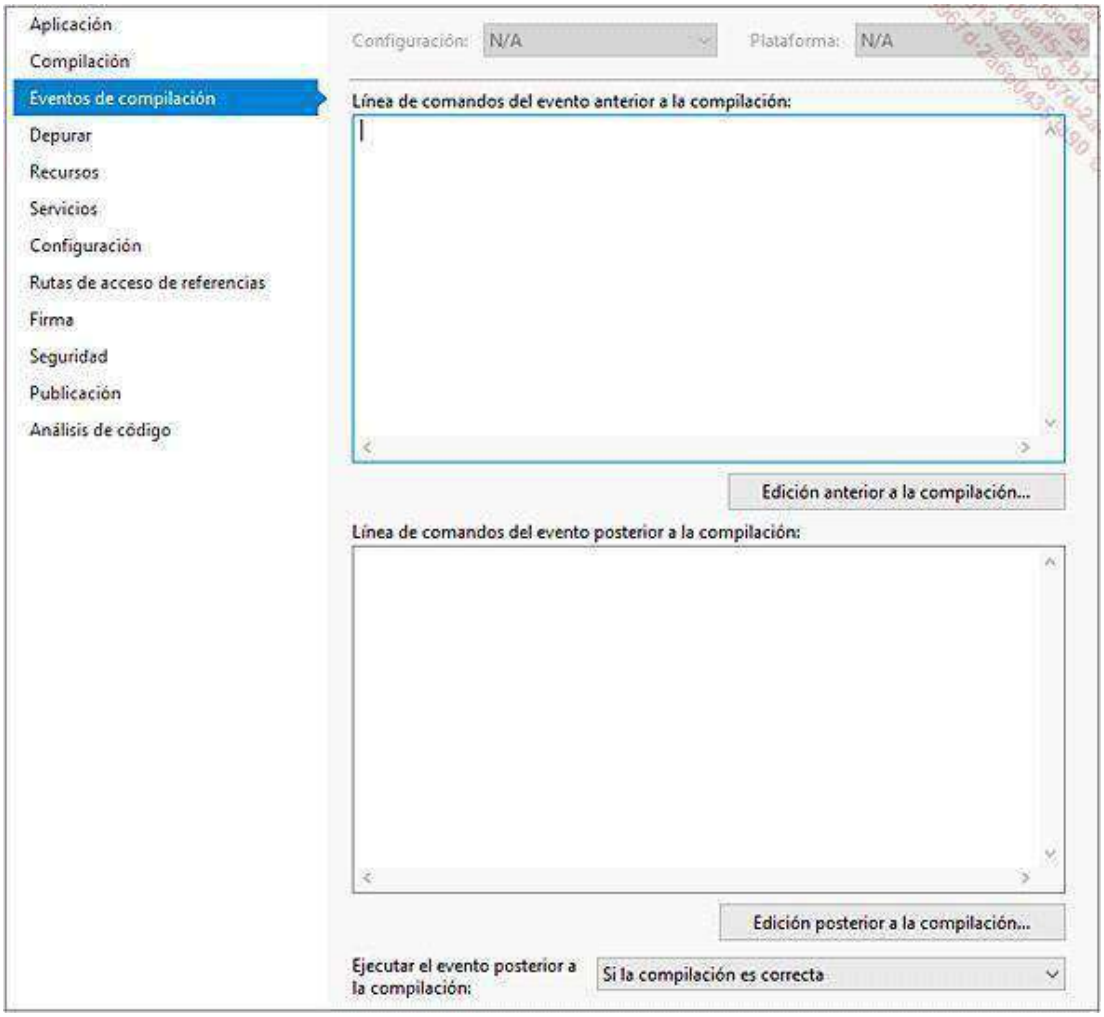
Esta opción indica al compilador que debe generarse una librería de clases de manera que sea compatible con el entorno COM.

Generar ensamblados de serialización

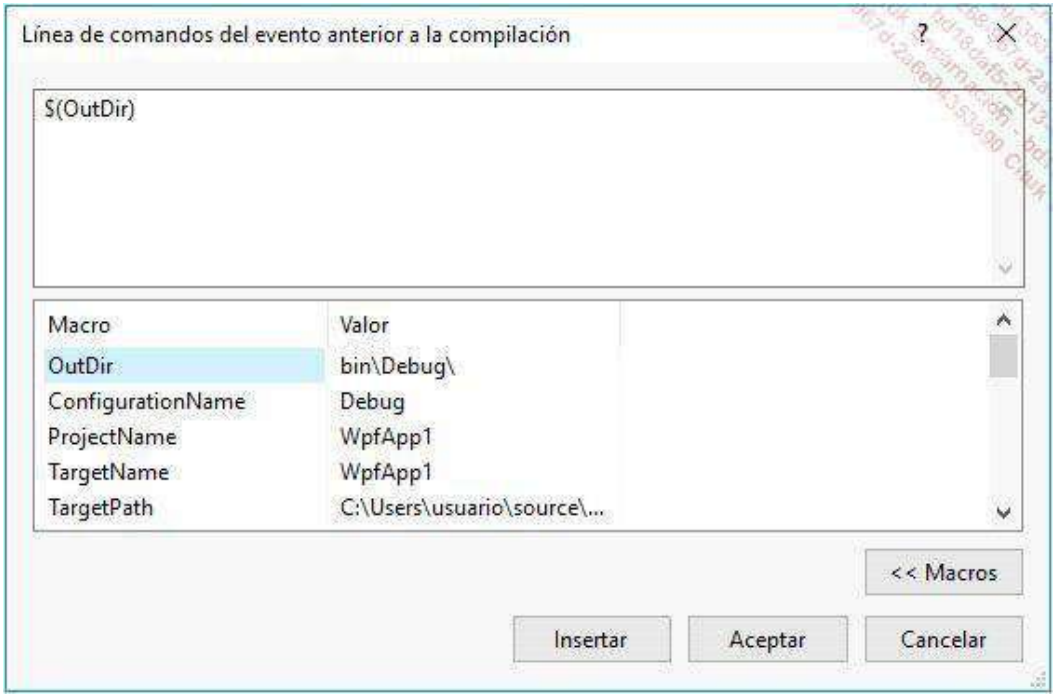
Indica al compilador el comportamiento que debe adoptarse de cara a las operaciones de serialización y deserialización de los objetos del proyecto.

c. Eventos de compilación

Los eventos de compilación permiten ejecutar uno o varios comandos antes o después de la compilación del proyecto. Estos eventos se utilizan a menudo para copiar archivos en una carpeta del proyecto en curso o de otro proyecto.



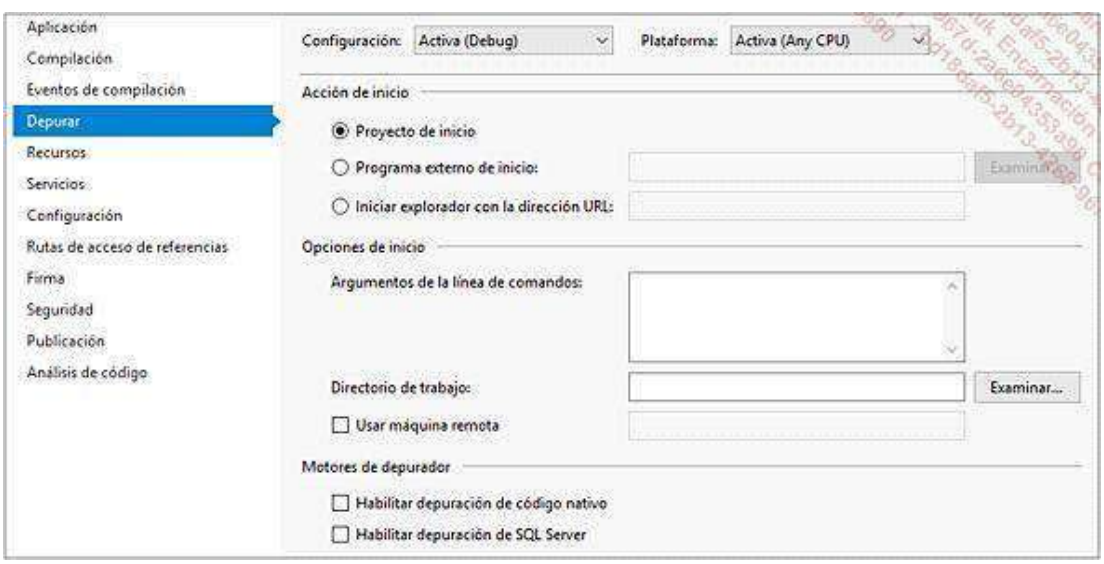
Estos comandos pueden introducirse también en las zonas de texto previstas para ello. Los botones **Edición anterior a la compilación** y **Edición posterior a la compilación** abren un cuadro de diálogo que ofrece macros suplementarias para simplificar la escritura de scripts.



Haciendo doble clic sobre una de las macros se inserta un valor especial en la línea de comandos. Este valor se evaluará cuando se produzca el evento de edición. La lista que contiene las macros permite previsualizar los valores de retorno.

d. Depurar

Esta página de propiedades permite modificar parámetros relativos a la ejecución de la aplicación generada por la compilación del proyecto.



Acción de inicio

Esta propiedad determina el comportamiento que adoptará Visual Studio tras la compilación de la aplicación y la ejecución de comandos posteriores a la compilación, si existen. Se proponen tres opciones:

- **Proyecto de inicio**: esta opción indica que el proyecto generado debe ejecutarse.

- **Programa externo de inicio**: esta opción ejecuta una aplicación que invoca al código generado y puede resultar particularmente útil para la depuración del código de una biblioteca de clases.
- **Iniciar explorador con la dirección URL**: esta última opción se utiliza para depurar aplicaciones web o aplicaciones de navegador WPF. Cuando se selecciona esta opción, Visual Studio abre el navegador por defecto de la máquina en la dirección indicada.

Opciones de inicio

Es posible pasar una lista de argumentos a la aplicación seleccionada como acción de inicio. Dichos argumentos deben incluirse en la zona **Argumentos de la línea de comandos**.

El **Directorio de trabajo**, que es por defecto la carpeta de la aplicación, también puede modificarse para ubicar la aplicación en un contexto particular.

Por último, la opción **Usar máquina remota** autoriza a depurar una aplicación ejecutándose sobre otro equipo. Cuando se selecciona esta opción es preciso indicar el nombre del equipo remoto.

Motores de depurador

Las distintas opciones propuestas permiten habilitar el uso de depuradores suplementarios como complemento al depurador integrado en Visual Studio.

e. Recursos

Los recursos permiten externalizar ciertos elementos de una aplicación. Estos elementos pueden ser cadenas de caracteres, imágenes o cualquier otro tipo de dato que pueda almacenarse fuera del código fuente para simplificar su búsqueda y eventualmente su modificación. Un mismo recurso puede utilizarse en varios lugares, lo que facilita el mantenimiento, evitando mantener datos duplicados.

El editor de recursos almacena los datos que se le proveen en el archivo **Resources.resx** que ha podido verse en las distintas plantillas de proyecto.

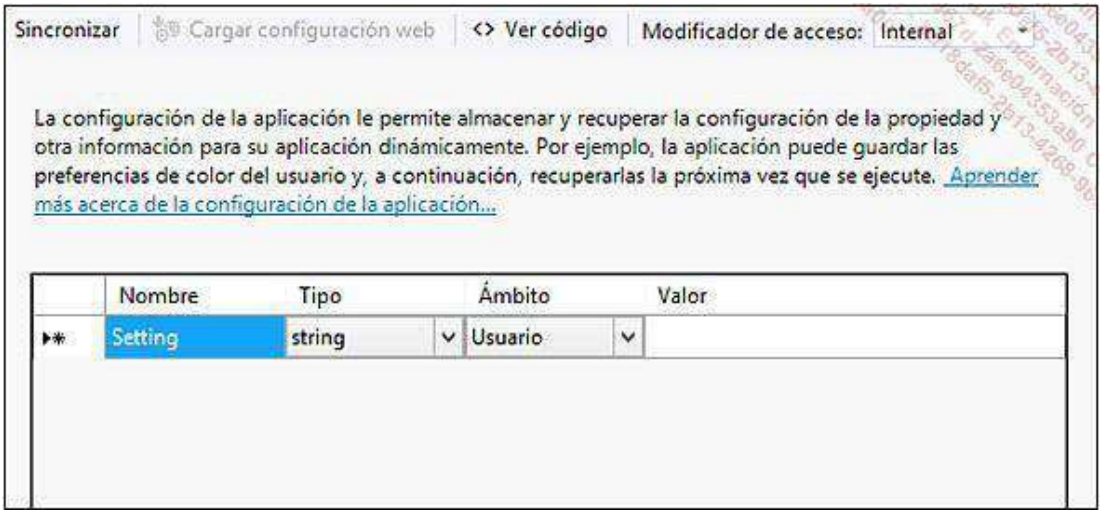
Cada recurso se identifica mediante un nombre, que se utilizará en el código para recuperar el valor asociado.

El recurso `NombreAplicacion` definido más arriba podrá utilizarse en el código de la siguiente manera:

```
string app = WpfApplication1.Properties.Resources.NombreAplicacion;
```

f. Parámetros

Los parámetros de la aplicación tienen una utilidad parecida a los recursos, pero se corresponden con el archivo **Settings.settings**. Permiten almacenar parámetros relativos a una aplicación o al usuario de una aplicación. Se utilizan habitualmente para almacenar las preferencias de los usuarios, por ejemplo tamaños de los tipos de letra, elementos que se desean mostrar u ocultar, etc.



Para cada parámetro es necesario facilitar un nombre que lo identificará de manera única, un tipo de datos, así como un ámbito y un valor. El ámbito puede seleccionarse de entre los siguientes valores:

- **Aplicación**: el parámetro es de solo lectura durante el uso de la aplicación.
- **Usuario**: el parámetro puede modificarse en tiempo de ejecución.

A diferencia de los recursos, los parámetros no se cargan automáticamente. Para realizar esta operación, se utiliza la propiedad `Default` del objeto `Settings`.

```
WpfApplication1.Properties.Settings.Default.Reload();
```

A continuación, es posible acceder al valor del parámetro de la misma manera que se hace con los recursos:

```
//En escritura
WpfApplication1.Properties.Settings.TamañoLetra = 14;
//En lectura
int tamañoletra =
WpfApplication1.Properties.Settings.TamañoLetra;
```

Por último, los parámetros modificados pueden salvaguardarse de la siguiente manera:

```
WpfApplication1.Properties.Settings.Default.Save();
```

Introducción

Como todos los lenguajes de programación, C# impone ciertas reglas al desarrollador. Estas reglas se ponen de manifiesto a través de la sintaxis del lenguaje, aunque cubren el amplio espectro funcional propuesto por C#. Antes de explorar en profundidad las funcionalidades del lenguaje en el siguiente capítulo, estudiaremos las nociones esenciales y fundamentales de C#: la creación de los datos que puede utilizar una aplicación y el procesamiento de estos datos.

Las variables

Los datos que se utilizan en un programa C# se representan mediante variables. Una variable es un espacio de memoria reservado al que se asigna arbitrariamente un nombre y cuyo contenido es un valor con un tipo fijado. Es posible manipular ese contenido en el código utilizando el nombre de la variable.

1. Nomenclatura de las variables

La especificación del lenguaje C# establece algunas reglas que deben tenerse en cuenta cuando se asigna el nombre a una variable:

- El nombre de una variable puede contener únicamente cifras, caracteres del alfabeto latino -acentuados o no-, el carácter ç o los caracteres especiales _ y µ.
- El nombre de una variable no puede, en ningún caso, empezar por una cifra. No obstante, puede incluirlas en cualquier otra posición.
- El lenguaje es sensible a mayúsculas y minúsculas: las variables `unavariable` y `unaVariable` son, por tanto, variables distintas.
- La longitud de un nombre de variable es prácticamente ilimitada: ¡el compilador no muestra ningún error hasta que tenga 30.000 caracteres! Se recomienda no utilizar nombres de variables demasiado largos, la longitud máxima, en la práctica, no suele superar un máximo de treinta caracteres.
- El nombre de una variable no puede ser una palabra clave del lenguaje. Es posible, no obstante, prefijar una palabra clave con un carácter autorizado o con el carácter @ para utilizar un nombre de variable similar.

Los nombres de variables siguientes son válidos en el compilador C#:

- `miVariable`
- `miVariableNúmero1`
- `@void` (`void` es una palabra clave de C#)
- `µn4_VàRïãbl3`

De manera general, es preferible utilizar nombres de variables explícitos, es decir, que permitan saber a qué se corresponde el valor almacenado en la variable, como `nombreCliente`, `importeCompra` o `edadDelEmpleado`.

2. Tipo de las variables

Una de las características de C# es la noción de tipado estático: cada variable se corresponde con un tipo de datos y no puede cambiar. Además, este tipo debe poder determinarse en tiempo de compilación.

a. Tipos valor y tipos referencia

Los distintos tipos que pueden utilizarse en C# se dividen en dos familias: los tipos valor y los tipos referencia. Esta noción puede resultar algo desconcertante a primera vista, pues una variable representa exactamente un dato y, por tanto, un valor. Este concepto está, de hecho, vinculado con la manera en que se almacena la información en memoria.

Cuando se utiliza una variable de tipo valor, se accede directamente a la zona de memoria que almacena el dato. En el momento de la creación de una variable de tipo valor, se reserva una zona de memoria equivalente al tipo.

Cada byte de esta zona se inicializa automáticamente con el valor binario 00000000. Nuestra variable tendrá, a continuación, un valor igual a 0 como valor binario.

En el caso de una variable de tipo referencia, el comportamiento es diferente. La zona de memoria asignada a nuestra variable contiene una dirección de memoria en la que se almacena el dato. La dirección de memoria se inicializa con el valor especial `null`, que no apunta a nada, mientras que la zona de memoria que contiene los datos no se ha inicializado. Se inicializará cuando se instancie la variable. Del mismo modo, la dirección de memoria almacenada en nuestra variable se actualizará.

Una variable de tipo referencia podrá, entonces, no contener ningún dato, mientras que una variable de tipo valor tendrá obligatoriamente un valor correspondiente a una secuencia de 0 binaria.

Esta diferencia en el comportamiento tiene una consecuencia importante: la copia de la variable se realiza por valor o por referencia, lo que significa que las variables de tipo valor se copiarán mientras que en las variables de tipo referencia se copiará la dirección que contiene la variable y será posible actuar sobre el dato real desde cualquiera de las variables que apunten sobre el mismo espacio de memoria.

b. Tipos integrados

El framework .NET incluye miles de tipos diferentes que pueden reutilizarse en los desarrollos. Entre estos tipos tenemos una quincena que podríamos considerar como fundamentales: son los tipos integrados (también llamados tipos primitivos). Son los tipos básicos sobre los que se construyen los demás tipos de la BCL así como los que crea el desarrollador en su propio código. Permiten definir variables que contienen datos muy simples.

Estos tipos tienen la particularidad de que disponen, cada uno, de un alias integrado en C#.

Tipos numéricos

Estos tipos permiten definir variables numéricas enteras o decimales. Cubren rangos de valores distintos y cada uno tiene una precisión específica. Algunos tipos estarán mejor adaptados para cálculos enteros, otros para cálculos en los que la precisión decimal resulte importante, como por ejemplo los cálculos financieros.

A continuación se enumeran los distintos tipos numéricos con sus alias así como los rangos de valores que cubren.

Tipo	Alias C#	Rango de valores cubierto	Tamaño en memoria
`System.Byte`	`byte`	0 a 255	1 byte
`System.SByte`	`sbyte`	-128 a 127	1 byte
`System.Int16`	`short`	-32768 a 32767	2 bytes
`System.UInt16`	`ushort`	0 a 65535	2 bytes
`System.Int32`	`int`	-2147483648 a 2147483647	4 bytes
`System.UInt32`	`uint`	0 a 4294967295	4 bytes
`System.Int64`	`long`	-9223372036854775808 a 9223372036854775807	8 bytes
`System.UInt64`	`ulong`	0 a 18446744073709551615	8 bytes
`System.Single`	`float`	±1,5e-45 a ±3,4e38	4 bytes
`System.Double`	`double`	±5,0e-324 a ±1,7e308	8 bytes
`System.Decimal`	`decimal`	±1,0e-28 a ±7,9e28	16 bytes

Los tipos numéricos primitivos son todos de tipo valor. Una variable de tipo numérico y no inicializada por el desarrollador tendrá un valor por defecto igual a 0.

.NET 4 incluye el tipo `System.Numerics.BigInteger` que permite manipular valores enteros de un tamaño arbitrario. Este tipo es también de tipo valor, pero no forma parte de los tipos integrados.

Los valores numéricos que se utilizan en el código se pueden definir a partir de tres bases numéricas:

- **Decimal** (base 10): es la base numérica que se utiliza por defecto.

  Por ejemplo: 280514

- **Hexadecimal** (base 16): se utiliza mucho en el mundo Web para la definición de colores, y más generalmente para codificar valores numéricos en un formato más comprimido. **Un valor hexadecimal se escribe con el prefijo 0x**.

  Por ejemplo: 0x447C2

- **Binario** (base 2): permite acercarse más al modo en que la máquina interpreta el código compilado. El uso de binarios en C# puede resultar valioso en el ámbito de las optimizaciones (compensaciones de bits en lugar de multiplicaciones por 2, almacenamiento de múltiples datos en una misma variable, etc.). **Los valores binarios se escriben con el prefijo 0b**.

  Por ejemplo: 0b01000100011111000010

La lectura de valores numéricos puede ser difícil en algunos casos, sobre todo en los que se representan en formato binario. El equipo encargado del desarrollo de C# ha integrado, en la séptima versión del lenguaje, la posibilidad de agregar separadores en los valores numéricos para mejorar la legibilidad global. El carácter utilizado para ello es:

- 280_514
- 0x04_47_C2
- 0b0100_0100_0111_1100_0010

Tipos textuales

Existen, en la BCL, dos tipos que permiten manipular caracteres Unicode y cadenas de caracteres Unicode: `System.Char` y `System.String`. Estos tipos tienen, respectivamente, los alias `char` y `string`.

El tipo `char` es un tipo valor que encapsula los mecanismos necesarios para procesar un carácter Unicode. Por ello, una variable de tipo `char` se almacena en memoria utilizando dos bytes.

Los valores de tipo `char` deben delimitarse por los caracteres `' '`: `'a'`.

Algunos caracteres tienen un significado particular para el lenguaje y deben utilizarse, obligatoriamente, con el carácter de escape \ para que se interpreten correctamente. Existen otros caracteres que no tienen representación gráfica y deben declararse utilizando secuencias específicas. La siguiente tabla resume las secuencias que pueden utilizarse.

Secuencia de escape	Carácter asociado
\'	Comilla simple '
\"	Comilla doble "
\\	Backslash \

\a	Alerta sonora
\b	Marcha atrás
\f	Salto de página
\n	Salto de línea
\r	Retorno de carro
\t	Tabulación horizontal
\v	Tabulación vertical
\0	Carácter nulo
\uXXXX	Carácter Unicode cuyo código hexadécimal es XXXX

El tipo `string` se utiliza para manipular tablas de objetos de tipo `char` y permite procesar cadenas que pueden representar hasta 4 GB, es decir 2 147 483 648 caracteres. A diferencia de los tipos integrados que hemos visto hasta ahora, el **tipo string es un tipo referencia**, lo que significa que una variable de este tipo puede tener el valor `null`.

Una cadena de caracteres se escribe entre los caracteres " ": "`una cadena de caracteres simpática pero algo larga`". Como ocurre con los `char`, algunos caracteres requieren utilizar una secuencia de escape.

Cabe destacar que las cadenas de caracteres son invariables, es decir, no pueden modificarse. Afortunadamente, el framework nos permite realizarlo, pero con un coste. Cada modificación que se realiza sobre una cadena de caracteres supone la creación de una nueva cadena que incluye la modificación. Este comportamiento, transparente para el desarrollador, puede suponer problemas de rendimiento cuando se trata de procesamientos importantes sobre este tipo de objetos. Es necesario, por tanto, conocer esta sutileza para evitar ese tipo de problemas.

Tipo booleano

Un valor booleano representa un estado verdadero o falso. Se trata de un tipo esencial en el desarrollo pues nos permite realizar verificaciones de condiciones.

Los valores booleanos "verdadero" y "falso" se escriben, respectivamente, `true` y `false`.

Arrays

Una variable de tipo array es una variable que contiene varios elementos del mismo tipo. Es posible acceder a cada uno de sus elementos utilizando un índice entero superior o igual a 0. El primer valor almacenado en el array tiene siempre el índice 0.

Un array puede tener varias dimensiones. En general, resulta difícil utilizar un array de más de tres dimensiones, pues no es evidente representar un objeto en cuatro o cinco dimensiones en un espacio de tres dimensiones.

La declaración de una variable de tipo array se realiza poniendo como sufijo al nombre de un tipo de datos el operador "corchetes": `[]`.

```
int[] array;
```

La variable `array` aquí declarada no puede utilizarse en este estado. En efecto, los arrays son tipos por referencia. Sin inicializarla, la variable tiene el valor `null`.

La inicialización de nuestro array se realiza mediante la palabra clave `new` e indicando el número máximo de elementos que puede contener.

```
array = new int[10];
```

Aquí, nuestro array puede contener diez elementos, indexados de 0 a 9.

También es posible inicializar una tabla proporcionando directamente los valores que debe contener. En este caso no es necesario indicar la longitud del array, se deduce directamente del número de elementos que se pasan durante su inicialización.

```
array = new int[] { 1 , 7, 123, 201 }
```

3. Declaración de variables

La primera etapa en el uso de una variable es su declaración. Si una variable se utiliza sin haber sido declarada, el compilador devuelve un error. La declaración del array anterior muestra cómo se asocia una variable a un tipo. A continuación, detallaremos la sintaxis de declaración de una variable.

La sintaxis general de declaración de una variable es la siguiente:

```
<Tipo de la variable> <Nombre de la variable> [ = valor inicial]
[, Nombre de una segunda variable];
```

Se indica el valor de la variable mediante el operador de asignación =. Si el valor inicial de la variable no se define en su declaración, la variable tendrá como valor el valor por defecto correspondiente a su tipo.

Como hemos visto antes, para un tipo referencia la variable tendrá un valor `null`, mientras que para un tipo valor, el valor se corresponderá con una inicialización de la variable en memoria con una secuencia de 0 binarios. Para un tipo numérico, la representación de este 0 binario será 0, para un valor booleano será `false`, para un char será el carácter nulo (`'\0'`).

```
System.String miCadenaDeCaracteres = "Esto es una cadena de
caracteres";
char carácter = 'a';

int numeroEntero1 = 4,
    numeroEntero2 = 128,
    numeroEntero3;
```

4. Ámbito de las variables

Hablamos de ámbito de una variable cuando queremos saber en qué porción de código es posible utilizar una variable. En C#, se corresponde con el bloque de código (entre los caracteres { y }) en el que se ha declarado.

Este bloque de código puede ser una clase, una función, una estructura de control en el interior de una función o incluso, aunque en la práctica no se utiliza, entre dos llaves posicionadas arbitrariamente en una función. Conviene conocer este último comportamiento, pues nadie está a salvo de un error en la manipulación y eliminar una línea puede ponernos en esa situación.

En el interior de una función, una variable puede utilizarse únicamente a partir de la línea en la que se ha declarado.

Una variable declarada en el interior de una clase puede, también, en función de los modificadores de acceso utilizados en su declaración, utilizarse fuera de la clase.

5. Modificadores de acceso

Los modificadores de acceso sobre las variables declaradas en una clase permiten determinar la visibilidad de las variables en el exterior de la clase.

Los distintos modificadores que podemos utilizar son, desde el más al menos restrictivo:

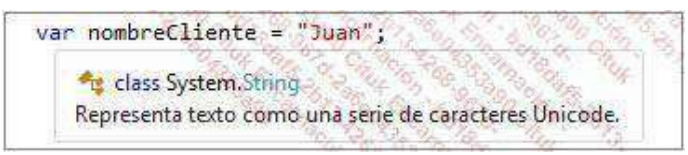
- `private`: la variable puede utilizarse únicamente en el interior de la clase en la que se ha declarado.
- `protected`: la variable puede utilizarse en el interior de la clase en la que se ha declarado así como en todas sus clases hijas.
- `internal`: la variable puede utilizarse en todas las clases del ensamblado que contiene la declaración de la variable.
- `protected internal`: la variable puede utilizarse en todas las clases del ensamblado que contiene su declaración así como en todas las clases hijas de la clase a la que pertenece.
- `public`: la variable puede utilizarse sin restricción alguna.

6. La palabra clave var y la inferencia de tipo

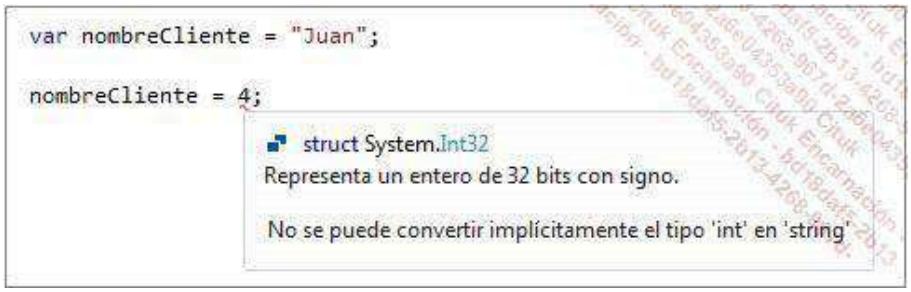
La inferencia de tipo es la capacidad que tiene el compilador de C# de determinar el tipo de una variable en función del valor que se le asigna. Se utiliza esta funcionalidad del lenguaje mediante la palabra clave `var` en lugar del tipo en la declaración de una variable.

```
var nombreCliente = "Juan";
```

Aquí, el compilador determina automáticamente que el tipo de `nombreCliente` es `string`:



Puede resultar tentador pensar que el tipo de nuestra variable es dinámico, dado que no lo hemos definido explícitamente. El tipo se fija, por tanto, de la misma manera que si hubiéramos especificado explícitamente el tipo `string`. El compilador rechaza, no obstante, asignar un valor de otro tipo a nuestra variable.



Para utilizar la inferencia de tipo, deben respetarse dos reglas:

- La variable debe ser local, es decir, debe estar declarada en el interior de una función.
- Es obligatorio asignar un valor a la variable en el momento de su declaración. En caso contrario, el compilador no puede determinar el tipo de la variable y devuelve un error.

Las constantes

Cuando se está programando una aplicación, llega un momento en el que se desea utilizar determinados valores que no sufren modificaciones. Un ejemplo típico es la constante matemática π. Este valor se define como se muestra a continuación en la clase `Math` del framework .NET:

```
public const double PI = 3.14159;
```

Esta instrucción es una declaración de variable a la que se le agrega la palabra clave `const`. El uso de esta palabra clave permite fijar el valor de `PI`.

Es posible definir constantes calculadas a partir de otras constantes:

```
public const double 2PI = 2 * Math.PI;
```

Los operadores

Los operadores son palabras clave del lenguaje que permiten ejecutar procesamientos sobre variables, valores de retorno de funciones o valores literales (los operandos).

La combinación de operadores y de operandos constituye una expresión que se evaluará en tiempo de ejecución y devolverá un resultado en función de los distintos elementos que la componen.

Ya hemos visto el operador de asignación =, que permite asignar un valor a una variable. Este valor puede ser otra variable, un valor literal, el resultado de una función o el resultado de una expresión.

Los distintos operadores pueden clasificarse en seis familias, relativas al tipo de procesamiento que permiten realizar.

1. Los operadores de acceso

El lenguaje C# define varios modos de acceso a los datos de los objetos que permite manipular. Cada uno de estos modos utiliza un operador particular.

a. Acceso simple: .

C# utiliza el símbolo . (punto) para permitir acceder a los miembros de un objeto, o eventualmente de un tipo.

```
//Aquí, se utiliza el operador . para acceder
//al miembro "Apellido" de la variable "persona"

string apellidoAlumno = persona.Apellido;
```

b. Acceso indexado: [ ]

El operador de acceso indexado ofrece la posibilidad de acceder a un valor contenido en una variable proporcionando un índice. El uso de variables de tipo array implica, por lo general, el uso de este operador para la lectura o la modificación de los datos que contiene.

```
int[] array = new int[10];
array[0] = 123 ;
```

c. Acceso con nulidad condicional: ?

En C#, el acceso a un miembro de un objeto cuyo valor es `null` provoca un error de ejecución.

```
//Si la variable persona vale null,
//se produce un error de ejecución

string apellidoAlumno = persona.Apellido;
```

A menudo sucede que el código que accede a un miembro "peligroso" se encapsula en un bloque que comprueba el valor del objeto principal antes de acceder a su miembro.

Esta técnica puede resultar algo pesada, en particular cuando es necesario anidar varias comprobaciones para acceder a un dato. Una de las funcionalidades que aportó C# 6 fue la posibilidad de utilizar el operador `?` para simplificar la escritura de este tipo de código. Puede, en efecto, combinarse con el operador de acceso simple o con el operador de acceso indexado para devolver el valor `null` si el objeto "padre" vale `null`.

```
string apellidoAlumno = persona?.Apellido;
```

En este ejemplo, si la variable `persona` está inicializada, la variable `apellido Alumno` contendrá el valor de `persona.Apellido`. En cambio, si la variable `persona` vale `null`, no se producirá ningún error de ejecución y la variable `apellidoAlumno` tendrá el valor `null`.

2. Los operadores aritméticos

Los operadores aritméticos permiten realizar cálculos sobre sus operandos.

Operador	Nombre de la operación	Ejemplo	Resultado
+	Suma	1 + 2	3
-	Resta	1 - 2	-1
*	Multiplicación	1 * 2	2
/	División	1 / 2	0.5
%	Módulo (resto de la división entera)	1 % 2	1

Estos operadores se utilizan principalmente con operandos numéricos. Ciertos tipos modifican el comportamiento de estos operadores para ejecutar procesamientos más acordes con su lógica. Es el caso particular del tipo `string` para el que el operador de suma permite realizar una concatenación, es decir, unir dos cadenas de caracteres.

3. Los operadores de comparación

Los operadores de comparación permiten definir expresiones cuyo resultado es un valor booleano. Se utilizan principalmente para evaluar condiciones en las estructuras de control.

Operador	Nombre del operador	Ejemplo	Resultado
==	Igualdad	`10 == 20`	`false`
!=	Desigualdad	`"C#" != "VB.NET"`	`true`
>	Superioridad	`10 > 20`	`false`
>=	Superioridad o igualdad	`10 >= 20`	`false`
<	Inferioridad	`10 < 20`	`true`
<=	Inferioridad o igualdad	`10 <= 20`	`true`
is	Comparación de tipo	`"Hello world" is string`	`true`

4. Los operadores condicionales

Existen dos operadores del lenguaje que permiten simplificar la escritura del código permitiendo eliminar bloques de

código vinculados a procesamientos condicionales.

a. Operador ternario: ? ... :

El operador ternario devuelve un valor en función de una expresión booleana. Recibe, para ello, tres operandos que son, por orden:

- Una expresión booleana.
- Una expresión cuyo valor se devolverá si la expresión booleana vale `true`.
- Una expresión cuyo valor se devolverá cuando la expresión booleana valga `false`.

El tipo de dato devuelto por las dos últimas expresiones debe ser idéntico. En caso contrario, el compilador generará un error de compilación.

El formato de una expresión que utiliza el operador ternario es el siguiente:

```
<expresión booleana> ? <expresión a evaluar si true> :
<expresión a evaluar si false>
```

La asignación de una cadena de caracteres en función de un número puede realizarse mediante el operador ternario de la siguiente manera:

```
int numClientes = 98;
string etiqueta = numClientes <= 100 ? numClientes.ToString() +
"clientes" : "Más de 100 clientes";

Console.WriteLine(etiqueta);     //Muestra "98 clientes"

numClientes = 120;
etiqueta = numClientes <= 100 ? numClientes.ToString() + "clientes" :
"Más de 100 clientes";

Console.WriteLine(etiqueta);     //Muestra "Más de 100 clientes"
```

b. Operador de fusión de valor nulo: ??

Este operador recibe dos operandos y devuelve el primero de ellos cuyo valor no sea `null`.

```
string ciudad = null;
string etiqueta = ciudad ?? "indefinida";

Console.WriteLine(etiqueta);     //Muestra  "indefinida"

ciudad = "VALENCIA";
etiqueta = ciudad ?? "indefinida";

Console.WriteLine(etiqueta);     //Muestra "VALENCIA"
```

5. Los operadores lógicos

Salvo el operador !, las operaciones lógicas permiten combinar varias expresiones que devuelven booleanos, como expresiones de comparación.

a. Negación: !

El operador de negación permite invertir un valor booleano. Este valor puede representarse mediante un literal, una variable o una expresión que devuelve un valor booleano.

```
bool booleano1 = true;
bool booleano2 = !booleano1;
//Llegados a este punto, booleano2 vale  false

bool comparación = !(booleano1 ==  booleano2);
// booleano1 == booleano2 devuelve false, y comparación vale true
```

b. Y lógico: &

El operador Y lógico permite combinar dos expresiones para determinar si ambas son verdaderas.

```
bool booleano1 = true;
bool booleano2 = false;

bool comparación1 = booleano1 & booleano2;
// comparación1 vale false

booleano2= true;
bool comparación2 = booleano1 & booleano2;
// comparación2 vale true
```

c. O lógico: |

El operador O lógico permite combinar dos expresiones para determinar si al menos una de ambas expresiones es verdadera.

```
bool booleano1 = true;
bool booleano2 = false;

bool comparación = booleano1 | booleano2;
// comparación vale true
```

d. O exclusivo: ^

El operador O exclusivo permite combinar dos expresiones para determinar si solamente una de las dos expresiones es verdadera.

```
bool booleano1 = true;
bool booleano2 = false;

bool comparación1 = booleano1 ^ booleano2;
// comparación1 vale true

booleano2 = true;
bool comparación2 = booleano1 ^ booleano2;
// comparación2 vale false
```

e. Y condicional: &&

El operador Y condicional permite combinar dos expresiones para determinar si ambas son verdaderas. El

segundo operando que se pasa a este operador no se evaluará si el primero se evalúa como falso.

```
bool booleano1 = false;
bool booleano2 = true;

bool comparación1 = booleano1 && booleano2;
// comparación1 vale false, y booleano2 no se evalúa

booleano1 = true;
bool comparación2 = booleano1 && booleano2;
// comparación2 vale true
```

f. O condicional: ||

El operador O condicional permite combinar dos expresiones para determinar si al menos una de ambas es verdadera.

El segundo operando que se pasa a este operador no se evaluará si el primero se evalúa como verdadero.

```
bool booleano1 = true;
bool booleano2 = true;

bool comparación1 = booleano1 || booleano2;
// comparación1 vale true, y booleano2 no se evalúa

booleano1 = false;
bool comparación2 = booleano1 || booleano2;
// comparación2 vale true
```

6. Los operadores binarios

Los operadores binarios no pueden aplicarse más que sobre tipos numéricos enteros. Realizan sobre sus operandos operaciones lógicas a nivel de bit.

a. Y binario: &

El operador Y binario realiza un Y lógico sobre cada uno de los bits de los operandos. El resultado de la expresión es un valor de tipo `int`. Realicemos la siguiente operación:

```
int resultado = 21 & 57;
```

	Operando 1	Operando 2	Resultado
Base 10	21	57	17
Base 2 (valor binario)	00010101	00111001	00010001

b. O binario: |

El operador O binario realiza un O lógico sobre cada uno de los bits de los operandos. El resultado de la expresión es un valor de tipo `int`. Realicemos la siguiente operación:

```
int resultado = 21 | 57;
```

	Operando 1	Operando 2	Resultado
Base 10	21	57	61
Base 2 (valor binario)	00010101	00111001	00111101

c. O exclusivo: ^

El operador O exclusivo binario realiza un O exclusivo sobre cada uno de los bits de los operandos. El resultado de la expresión es un valor de tipo `int`.

	Operando 1	Operando 2	Resultado
Base 10	21	57	44
Base 2 (valor binario)	00010101	00111001	00101100

d. Negación: ~

El operador de Negación binario invierte el valor de cada uno de los bits de su operando. Realicemos la siguiente operación:

```
int resultado = ~21;
```

	Operando	Resultado
Base 10	21	-22
Base 2 (valor binario)	00010101	11111111111111111111111111101010

El operador de negación devuelve un valor del mismo tipo que su operando: aquí, `int`. Este tipo se codifica en 32 bits, lo que explica la longitud del resultado devuelto.

e. Desplazar a la derecha: >>

Este operador desplaza a la derecha los bits que componen el primer operando un número de posiciones especificado por el segundo operando. Realicemos la siguiente operación:

```
int resultado = 21 >> 2;
```

	Operando 1	Operando 2	Resultado
Base 10	21	2	5
Base 2 (valor binario)	00010101		00000101

f. Desplazar a la izquierda: <<

Este operador desplaza a la izquierda los bits que componen el primer operando un número de posiciones especificado por el segundo operando. Realicemos la siguiente operación:

```
int resultado = 21 << 2;
```

	Operando 1	Operando 2	Resultado
Base 10	21	2	84
Base 2 (valor binario)	00010101		01010100

Las estructuras de control

Las estructuras de control permiten crear un flujo de procesamiento complejo. Existen dos tipos de estructuras de control:

- Las estructuras condicionales, que ejecutan un procesamiento en función de una condición.
- Las estructuras de bucles, que ejecutan varias veces un mismo procesamiento.

1. Las estructuras condicionales

Existen dos estructuras condicionales.

a. if ... else

La estructura `if ... else` recibe como parámetro una expresión que devuelve un valor booleano, como una expresión de comparación. Si la condición se cumple, el bloque de código correspondiente se ejecuta.

En caso contrario, se ofrecen tres opciones:

- Se define un bloque `else` justo a continuación del `if` y en ese caso se ejecuta el código de este bloque.
- Existen varios bloques `else if` tras el `if` y cada expresión de condición se ejecuta hasta que una de ellas devuelve `true`. Entonces se ejecuta el bloque de la instrucción `else if` (o `else`, eventualmente).
- No existe ningún bloque `else` o `else if`, y la ejecución del código continúa tras la salida del bloque `if`.

La sintaxis de esta estructura es la siguiente:

```
if (<expresión de condición>)
{
    //Situar aquí el código a ejecutar si la expresión devuelve true
}
[ else if (<expresión de condición2>)
{
    //Situar aquí el código a ejecutar si la primera expresión
    //devuelve false y la segunda expresión devuelve true
}]
[else if...]
[ else
{
    //Situar aquí el código a ejecutar si ninguna de las expresiones
    //de los bloques if y else if devuelve true
}]
```

El uso de bloques `else if` o `else` es opcional. Pero, lógicamente, un bloque `else` o `else if` no puede existir sin un bloque `if`.

Esta estructura puede tener varias decenas de bloques `else if` asociados, pero no puede tener más que un único bloque `else`.

b. switch

La estructura `switch` selecciona una rama de ejecución en función de un argumento y la comprobación de los valores de ese parámetro. El argumento que se pasa a esta estructura debe ser de uno de los siguientes tipos:

- `bool` o `Nullable<bool>`
- `char` o `Nullable<char>`
- `string`
- Uno de los tipos numéricos enteros: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` o `ulong` o un tipo `Nullable` correspondiente (como `Nullable<int>`).
- Una enumeración (consulte el capítulo Programación orientada a objetos con C# - sección Las enumeraciones).

El cuerpo de esta estructura está compuesto por etiquetas `case` que permiten comprobar, cada una, un valor. También es posible definir una etiqueta `default` que gestione los casos que no se comprueban explícitamente.

La sintaxis de la estructura `switch` es la siguiente:

```
switch (<argumento>)
{
   case <valor1> :
          //situar aquí el procesamiento a ejecutar
          break;
   case <valor2>:
          //situar aquí el procesamiento a ejecutar
          break;
   [ case ... ]
   [ default :
          //situar aquí el procesamiento a ejecutar si nuestro
          //valor no se ha asociado a ningún case
          break; ]
}
```

En C#, a diferencia de otros lenguajes, como C++, es imposible ejecutar un procesamiento en una etiqueta y continuar ejecutando el código de la etiqueta siguiente. La palabra clave `break` sirve, aquí, para hacer este comportamiento explícito. Si alguna etiqueta que contiene los procesamientos no contiene una llamada a esta instrucción, el compilador devolverá un error.

En cambio, es posible realizar el mismo procesamiento dentro de dos etiquetas. Para ello basta con declarar ambas etiquetas directamente una a continuación de la otra.

A continuación se muestra un ejemplo de estructura `switch` que utiliza estos distintos comportamientos.

```
int miVariable = 4;

switch (miVariable)
{
    case 1:
        Console.WriteLine("¡Soy un 1!");
        break;
    case 2:
```

```
    case 3:
        Console.WriteLine("¡Soy un 2 o un 3!");
        break;
    default:
        Console.WriteLine("No soy ni un 1, ni un 2, ni un 3...
¿Qué seré?");
        break;
}
```

Esto producirá la siguiente salida:

```
No soy ni un 1, ni un 2, ni un 3... ¿Qué seré?
```

c. El pattern matching

La especificación de C# 7 incorpora un concepto del mundo de la programación funcional: coincidencia de patrones (pattern matching). Esta técnica permite probar si un valor tiene una cierta "forma" y extraer la información que tiene cuando finaliza la prueba. Su integración se basa en estructuras de control `if ... else` y `switch`.

Se pueden utilizar tres tipos de patrones:

- Los patrones constantes: de la forma `val`, donde `val` es una expresión constante que comprueba que si valores son iguales.
- Los patrones de tipo: de la forma `TType val`, donde `TType` es un tipo y `val` es una variable.
- Los patrones de asignación: de la forma `var val`, donde `val` es una variable.

El siguiente método utiliza cada uno de estos patrones en los bloques `if`.

```
static void Main(string[] args)
{
    object x = 6;
    //Patrones constantes
    if (x is 4) Console.WriteLine("x is 4");
    if (x is 6) Console.WriteLine("x is  6");
    //Patrones de tipo
    if (x is string s) Console.WriteLine("x is string s");
    if (x is int i) Console.WriteLine("x is int  i");
    //Patrones de asignación
    if (x is var v) Console.WriteLine("x is var v");
    Console.ReadLine();
}
```

Este código devuelve la siguiente salida:

```
x is 6
x is int i
x is var v
```

El siguiente código muestra el uso de la coincidencia de patrones dentro de una estructura de control `switch`.

```
static void Main(string[] args)
{
    Proceso(6);
    Console.WriteLine();
    Proceso(16);
    Console.WriteLine();
    Proceso("cadena");
    Console.WriteLine();
    string s = null;
    Proceso(s);
    Console.WriteLine();
    Proceso (new object[] { 1, 2, 3, 4 });
    Console.WriteLine();

    Console.ReadLine();
}
static void Proceso (object x)
{
    switch (x)
    {
        case string s:
            Console.WriteLine("case string s");
            break;
        case int i when (i > 10):
            Console.WriteLine("case int i when (i > 10)");
            break;
        case int i when (i < 10):
            Console.WriteLine("case int i when (i < 20)");
            break;
        default:
            Console.WriteLine("default");
            break;
        case null:
            Console.WriteLine("case null");
            break;
    }
}
```

La segunda ejecución del método `Proceso` muestra que el orden de los bloques `case` es importante. De hecho, aunque 16 es mayor que 10 y menor que 20, solo se ejecuta el primer caso.

La llamada a `Proceso(s)` no ha tratado el bloque `case string s`. El motivo es que, si bien se corresponde en cuanto al tipo, no se invoca. Los patrones tipo solos no se invocan para tratar un valor `null`, para evitar que un valor `null` pueda pasar en un bloque `case null`.

La ejecución de la llamada a `Proceso(s)`, según esta regla, se debería tratar en el bloque `default`. O ser tratada correctamente como `null`. De hecho, el comportamiento del bloque `default` se conserva: SIEMPRE se evalúa el último, independientemente de su posición en la estructura `switch`.

2. Las estructuras de iteración

A menudo es útil repetir el mismo procesamiento sobre varias variables, como los elementos de una tabla. Para ello, C# provee cuatro tipos de estructuras capaces de ejecutar en bucle un procesamiento.

a. for

La estructura `for` se utiliza cuando el número de iteraciones que se desea ejecutar se conoce antes de comenzar con la ejecución. Esta estructura recibe tres argumentos, separados por comas, que son:

- **Una instrucción de inicialización** que se ejecuta antes de empezar con las iteraciones. Esta instrucción es, generalmente, la inicialización de una variable "contador".
- **Una condición que se evalúa antes de cada iteración**. Si esta condición se evalúa como `false`, el bucle se detiene. Esta instrucción es, generalmente, una comparación de la variable "contador" con un valor igual a la longitud del array.
- **Una instrucción que se ejecuta al final de cada iteración** y antes de la evaluación de la condición de salida. Esta instrucción es, generalmente, una modificación de la variable "contador" incrementada o decrementada.

Estas tres instrucciones son opcionales, no obstante preste atención, pues se puede llegar a programar un bucle infinito.

La sintaxis para escribir bucles `for` es la siguiente:

```
for ([<instrucción de inicialización>]; [<condición de inicio
de la iteración>]; [<instrucción de fin de  iteración>])
{
    //Código que se debe ejecutar con cada iteración
}
```

Por regla general, el uso de un bucle `for` es:

```
int numeroIteraciones = 7;
for (int contador = 0; contador < numeroIteraciones; contador++)
{

    //Código que se debe ejecutar con cada iteración
}
```

La instrucción de final de iteración puede parecer extraña a primera vista. El operador `++` es el operador de incremento. El uso de la instrucción `contador++` es equivalente a la instrucción:

```
contador = contador + 1;
```

b. while

La estructura `while` se utiliza principalmente cuando el número de iteraciones se desconoce por completo antes de iniciar la ejecución del bucle. Recibe como parámetro una expresión booleana. El bucle termina su ejecución cuando esta condición devuelve el valor `false`.

Un bucle `while` se escribe de la siguiente manera:

```
while (<condición>)
{
   //Procesamiento a ejecutar
}
```

c. do ... while

Las estructuras `do ... while` se utilizan cuando se requiere que se ejecute, como mínimo una iteración, pero se desconoce el número total de iteraciones antes de comenzar con la ejecución.

Como los bucles `while`, reciben como parámetro una expresión booleana. Su sintaxis es la siguiente:

```
do
{
    //Procesamiento a ejecutar al menos una vez
} while (<condición>)
```

d. foreach

Los bucles `foreach` son estructuras algo particulares. Permiten iterar sobre los elementos de una colección. La colección utilizada debe respetar una condición: implementar la interfaz `IEnumerable`. Esta interfaz la implementan, generalmente, los arrays.


El concepto de interfaz se aborda en el capítulo Programación orientada a objetos con C# - sección Las interfaces.

La sintaxis de esta estructura es la siguiente:

```
foreach (<tipo> <nombre de la variable> in <colección>)
{
   //Procesamiento
}
```

La escritura del bucle `foreach` no es evidente, veamos un ejemplo algo más detallado.

```
string[] arcoIris = {
"rojo", "naranja", "amarillo", "verde", "azul", "añil", "violeta" };

foreach (string color in arcoIris)
{
    Console.WriteLine(color);
}
```

e. Controlar la ejecución de un bucle

Como hemos visto antes, es muy sencillo escribir un bucle infinito en C#. Para salir, basta con utilizar la palabra clave `break` que también hemos visto con la estructura `switch`.

```
int i = 0;
//Creación de un bucle infinito
while (true)
{
     //Condición de salida
     if (i == 5)
           break;
     i++;
}
```

Para detener el procesamiento de un bucle y pasar directamente a la iteración siguiente se utiliza la palabra clave `continue`.

```
for (int i = 0; i < 20; i++)
{
   if (i == 5)
         continue;
   //Procesamientos específicos para los números diferentes a 5
}
```

3. Otras estructuras

Existen otras dos estructuras en C#, la primera se utiliza mucho más que la segunda.

a. using

La estructura `using` es una ayuda sintáctica destinada al uso de objetos que implementan la interfaz `IDisposable`. Las clases que implementan esta interfaz utilizan, generalmente, recursos externos como ficheros, etc. Estos recursos se liberan mediante el método `Dispose`.

El tipo `FileStream` representa un flujo de datos vinculado a un archivo. Implementa la interfaz `IDisposable` de manera que permite liberar los recursos utilizados cuando finaliza el procesamiento.

Sin la instrucción `using`, el código que permite su uso es similar al siguiente:

```
FileStream stream = new FileStream(@"C:\miarchivo.txt",
FileMode.Open);

// Procesamientos...

stream.Dispose();
```

Este tipo de código está sujeto a problemas, pues no nos asegura que se invoca en todas las ocasiones al

método `Dispose`. El flujo de ejecución puede verse interrumpido por un error o puede que el desarrollador olvide realizar la llamada al método.

La reescritura del código con ayuda de un bloque `using` da el siguiente resultado:

```
using (FileStream stream = new FileStream(@"C:\miarchivo.txt",
FileMode.Open))
{
    //Procesamientos
}
```

La llamada al método `Dispose` la genera el compilador, de modo que se realiza sistemáticamente, incluso en el caso de producirse algún error en la ejecución. Además, es imposible para el desarrollador omitir esta llamada al método dado que es el compilador el que la gestiona.

Por estos motivos, el uso de estructuras `using` se considera como una buena práctica de C#.

b. goto

La estructura `goto` es una herencia de lenguajes como BASIC. Permite realizar saltos en la ejecución del código, desde el punto de ejecución en curso hasta una etiqueta predefinida. Dado que esto complica la lectura del código, esta estructura se utiliza muy poco en el código C#, aunque puede resultar muy práctica, en particular para salir de procesamientos muy complejos.

Se utiliza de la siguiente manera:

```
<nombre de la etiqueta>:
//Procesamientos
goto <nombre de la etiqueta>;
```

La instrucción `goto` también puede situarse antes de la definición de la etiqueta.

```
Console.WriteLine("Ejecución de código...");

goto etiqueta;
Console.WriteLine("entre el goto y la etiqueta");

etiqueta:
Console.WriteLine("tras definir la etiqueta");
```

Lo que produce la siguiente salida:

```
Ejecución de código...
tras definir la  etiqueta
```

Las funciones

Las funciones son un elemento central en el desarrollo con C#. En efecto, todas las instrucciones de una aplicación escrita en C# deben situarse en funciones.

Cada función representa una unidad de procesamiento reutilizable que puede recibir uno o varios parámetros y devolver un valor.

La escritura de funciones permite estructurar el código desacoplando de manera lógica las funcionalidades desarrolladas. Se recomienda, para una buena legibilidad y una buena mantenibilidad, limitar la longitud de las funciones.

Muchos desarrolladores optan, de este modo, por una longitud que permita ver el código completo de una función en "una pantalla". Esta longitud es del todo relativa, pero puede convenir a cada uno. Esta regla no es, evidentemente, absoluta, pero puede ayudar, especialmente en casos de trabajo en equipo o en la relectura y la depuración.

Para alcanzar este objetivo es necesario limitar la responsabilidad de las funciones: cada una debería realizar un tipo de tarea únicamente.

1. Escritura de una función

La sintaxis general para escribir una función es la siguiente:

```
<modificador de acceso> <tipo de retorno> <Nombre de la función>(
[parámetro 1, [parámetro 2]...])
{
    //Procesamientos
    return <valor>;
}
```

El par nombre/lista de parámetros define la firma de la función. Es lo que va a permitir al compilador reconocer la existencia de una función durante su llamada.

El tipo de retorno de la función puede ser cualquier tipo accesible por el código de la función.

Una función debe, obligatoriamente y en todos los casos, devolver un valor explícito. Para ello, se utiliza la palabra clave `return` seguida del valor a devolver.

```
public int CalcularEdadDelCapitán()
{
    int edadDelCapitán = 0;

    //Procesamientos...

    return edadDelCapitán;
}

public string RecuperarTipoDeDía()
{
    if (DateTime.Today.Day % 2 == 0)
            return "DIA PAR";
```

```
}
```

Aquí, la función `RecuperarTipoDeDía` no es válida. En efecto, no siempre devuelve un valor, salvo cuando el día es par. El compilador indica que existe un problema:

Es preciso corregir este método para que el compilador sepa interpretarlo:

```
public string RecuperarTipoDeDía()
{
    if (DateTime.Today.Day % 2 == 0)
            return "DIA PAR";
    //Aquí se gestionan los demás casos, es decir los días impares
    return "DIA IMPAR";
}
```

Esta vez, el proceso de compilación termina sin errores.

2. Parámetros de función

Es habitual tener que realizar procesamientos que dependen de datos particulares. Para ello, las funciones pueden recibir parámetros.

Los parámetros son variables locales a una función, pero cuyo valor se proporciona desde la llamada a esta función. Se declara un parámetro y se especifica su tipo y su nombre.

```
public decimal CalcularPrecioConIVA(decimal precioSinIVA)
{
    //Aplicamos un IVA del 21%
    return precioSinIVA *  1.21;
}
```

Es posible invocar nuestra función de la siguiente manera:

```
decimal precio = CalcularPrecioConIVA(10.25);
```

Argumentos opcionales

Es posible agregar parámetros opcionales a nuestra función, es decir, parámetros a los que el desarrollador podrá optar por asignar o no valor en el momento de la llamada a la función. Para utilizar esta funcionalidad, es obligatorio otorgar un valor por defecto a cada uno de los parámetros opcionales.

Los parámetros opcionales deben ser los últimos en declararse en la firma de la función.

Modifiquemos la función anterior de manera que nos permita recibir parámetros opcionales correspondientes al tipo de IVA y a un descuento que se aplicará sobre el precio con IVA:

```
public decimal CalcularPrecioConIVA(decimal precioSinIVA,
decimal tasaIVA = 0.21, decimal importeDescuento =  0)
{
    //Aplicamos el IVA y el descuento
    return precioSinIVA * (1 + tasaIVA) - importeDescuento;
}
```

A continuación, es posible invocar a esta función de las siguientes maneras:

```
//Aplicamos el IVA por defecto
decimal precio = CalcularPrecioConIVA(10.25);

//Aplicamos un IVA reducido del 10%
decimal precio2 = CalcularPrecioConIVA(10.25, 0.1);

//Aplicamos un IVA reducido del 10% y un descuento de 1
decimal precio3 = CalcularPrecioConIVA(10.25, 0.1,  1);
```

Argumentos con nombre

Para calcular un precio con IVA con una tasa del 21%, es decir el IVA por defecto, y un descuento específico, tenemos la posibilidad de invocar a la función `CalcularPrecioConIVA` indicando el valor de todos los parámetros. Pero esto induce cierta redundancia inútil a nivel del argumento opcional `tasaIVA`. Esta redundancia hará que sea difícil de mantener si el número de parámetros opcionales crece todavía más.

El uso de parámetros con nombre está indicado para resolver este problema.

Es posible, en efecto, especificar únicamente aquellos parámetros opcionales de los que se desea proveer un valor especificando el nombre de los parámetros junto a su valor.

Para la función `CalcularPrecioConIVA`, obtendríamos el siguiente código:

```
decimal precio = CalcularPrecioConIVA(10.25, importeDescuento: 1);
```

Es posible, también, especificar el nombre de todos los parámetros que se pasan a la función y modificar el orden de los mismos en la llamada, tal y como se muestra a continuación:

```
decimal precio = CalcularPrecioConIVA(tasaIVA: 0.1, importeDescuento:
1, precioSinIVA: 10,25);
```

Argumentos variables

Algunas funciones deben aceptar un número variable de parámetros. Esto es así especialmente en el caso de la función `Main` de una aplicación de consola.

En efecto, en el caso de una aplicación de consola, la función `Main` recibe como parámetros todos los argumentos que se facilitan a la aplicación. Es imposible saber de antemano cuántos argumentos se pasarán.

En este caso, se define en la función un parámetro particular de tipo array, declarado con la palabra clave `params`. Tras la llamada a la función, es posible especificar un número variable de valores que estarán agrupados en un único parámetro.

```
public int SumarVariosEnteros(params int[] enteros)
{
    int resultado = 0;
    for (int contador = 0; contador < enteros.Length; contador++)
    {
            resultado += enteros[contador];
    }
    return resultado;
}
```

Esta función se utiliza de las siguientes maneras:

```
int resultadoSuma = SumarVariosEnteros(1, 5, 123, 4, -5);
int resultadoSuma2 = SumarVariosEnteros(1, 5);
int resultadoSuma3 = SumarVariosEnteros();
```

Es posible definir un único parámetro variable por función. Es posible, no obstante, definir parámetros obligatorios antes del parámetro variable.

```
public string SumarNumerosEnterosAUnNumero(int unNumero,
params int[] enteros)
{
    int resultado = unNumero;
    for (int contador = 0; contador < enteros.Length; contador++)
    {
            resultado += enteros[contador];
    }
    return resultado;
}
```

Esta función se invoca de la misma manera que la anterior, pero es obligatorio pasar al menos un valor entero como parámetro durante su llamada.

Argumentos de escritura: out y ref

Algunos casos de uso pueden requerir modificar uno de los valores que se pasan como parámetro a una función. Por defecto, un parámetro de una función es de solo lectura siempre que su tipo sea un tipo valor. Para modificar este comportamiento, C# incluye las palabras clave `out` y `ref`. Si bien ambas permiten repercutir la modificación del valor de un parámetro en la función que invoca, no se utilizan en el mismo contexto ni de la misma manera.

La palabra clave `ref` delante del tipo de un parámetro indica que debe proveerse una **variable inicializada** en la llamada a la función, e indica a su vez que puede modificarse en el código ejecutado en la función. Es posible, por ejemplo, que la función del cálculo de precio con IVA que hemos visto antes modifique el valor del descuento aplicado, dado que es un aspecto importante.

```
public decimal CalcularPrecioConIVA(decimal precioSinIVA, decimal tasaIVA,
ref decimal importeDescuento)
{
    //Si el descuento es superior al 20% del precio sin IVA,
    //se limita el importe al 20% del precio
    if (importeDescuento > precioSinIVA *  0.2)
    {
        importeDescuento = precioSinIVA *  0.2;
    }
   //Aplicamos el IVA
   return precioSinIVA * (1 + tasaIVA) - importeDescuento;
}
```

Esta función se invoca mediante el siguiente código:

```
decimal importeDescuento = 4;
decimal precioConIVA = CalcularPrecioConIVA(40, 0.21, ref importeDescuento);
```

El código que invoca a la función debe utilizar obligatoriamente la palabra clave `ref` delante del nombre de la variable que se pasa como referencia y la variable debe, obligatoriamente, estar inicializada.

La palabra clave `out` se utiliza cuando una función debe devolver varios datos. Podríamos imaginar que la función `CalcularPrecioConIVA` pudiera devolver de manera clásica el precio con IVA de un artículo, además del porcentaje de descuento acordado relativo al importe sin IVA mediante un parámetro marcado con la palabra clave `out`.

```
public decimal CalcularPrecioConIVA(decimal precioSinIVA, decimal tasaIVA,
decimal importeDescuento, out decimal porcentajeDescuento)
{
    //Cálculo porcentaje de descuento acordado sobre el precio sin IVA
    porcentajeDescuento = importeDescuento * 100 /  precioSinIVA;

    //Aplicamos el IVA
    return precioSinIVA * (1 + tasaIVA) - importeDescuento;
}
```

En este caso, la función se invoca de la siguiente manera:

```
decimal porcentaje;
decimal precioConIVA = CalcularPrecioConIVA(40, 0.21, 5, out porcentaje);
```

Tras su ejecución, la variable porcentaje contiene el valor calculado en el interior de la función. Cabe destacar que

no está inicializada antes de la ejecución del cálculo como habría sido necesario si se tratase de una variable pasada mediante la palabra clave `ref`.

Para simplificar la escritura de código de llamada a un método utilizando uno (o varios) parámetros `out`, es posible declarar la variable de retorno directamente en los argumentos del método.

```
decimal precioConIVA = CalcularPrecioConIVA(40, 0.2, 5, out var porcentaje);
```

También es posible ignorar un parámetro `out` nombrándolo con un "espacio en blanco". Su valor no se devolverá al código que llama a la función.

```
decimal precioConIVA = CalcularPrecioConIVA(40, 0.2, 5, out _);
```

3. Procedimientos

Es posible escribir funciones que no devuelvan ningún valor. Estas funciones se denominan procedimientos.

En la práctica, se define que su tipo de retorno es `void`. El uso de esta palabra clave permite al compilador saber que no es posible situar un procedimiento en un lugar donde se espere un valor (una asignación o un cálculo, por ejemplo).

```
public void UnProcedimiento()
{
    //Procesamiento

    //Palabra clave opcional en un procedimiento
    return;
}
```

En un procedimiento, la palabra clave `return` es totalmente opcional. Puede resultar interesante, no obstante, utilizarla para acortar un proceso, sobre todo en función del contexto.

```
public void EncenderLaLuz()
{
    bool luzEncendida = false;
    if (luzEncendida)
            return;

    //Si se llega aquí, la luz está, necesariamente, encendida
    PulsarInterruptor();
    luzEncendida = true;
}
```

4. Sobrecargas

El lenguaje C# permite definir varias funciones o procedimientos con el mismo nombre pero con firmas diferentes, es

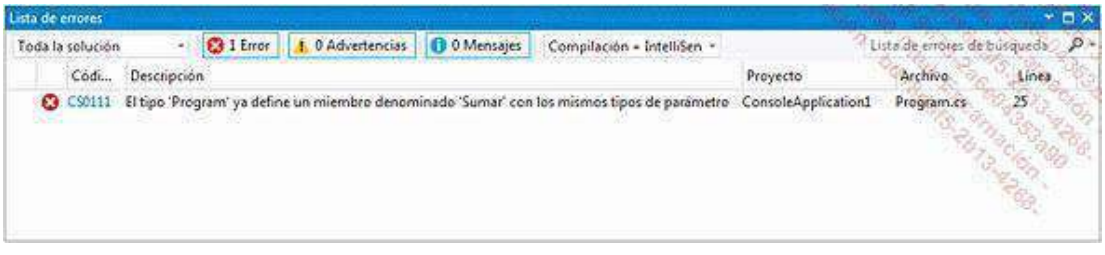
decir, funciones cuyos parámetros son diferentes de manera significativa: el número o el tipo de estos parámetros debe ser diferente. Estas funciones se denominan **sobrecargas**.

```
public int Sumar(int numero1, int numero2)
{
    return numero1 + numero2;
}

public int Sumar(int numero1, int numero2, int numero3)
{
    return numero1 + numero2 + numero3;
}
```

Estas dos funciones pueden coexistir sin problema: el compilador C# es capaz de saber qué función ejecutar según el número de parámetros que se pasan en la llamada.

```
public int Sumar(int numero1, int numero2, int numero3 = 3)
{
    return numero1 + numero2 + numero3;
}

public int Sumar(int numero1, int numero2, int numero3)
{
    return numero1 + numero2;
}
```

En este caso, se obtiene un error de compilación. En efecto, si se realiza la siguiente llamada, el compilador sería incapaz de saber a qué sobrecarga invocar.



En cambio, es posible escribir el siguiente código:

```
public int Sumar(int numero1, int numero2)
{
    return numero1 + numero2;
}

public decimal Sumar(decimal numero1, decimal numero2)
{
    return numero1 + numero2;
}
```

El compilador es capaz de saber que la primera función debe ejecutarse cuando se suman dos enteros, mientras que la segunda función está adaptada para sumar números decimales.

5. Funciones locales

Para simplificar un algoritmo o mejorar la legibilidad de un trozo de código, los desarrolladores frecuentemente tienen la necesidad de crear funciones de utilidades. En muchos casos, son específicas en un contexto concreto y no se pueden reutilizar. La conclusión final es clara: para ganar simplicidad, se añade "contaminación" al código.

C# 7 ha introducido un concepto, familiar para los desarrolladores de JavaScript, que elimina este tipo de contaminación: las funciones locales. La base de su implementación es extremadamente simple ya que consiste en anidar una función de utilidad (o varias) en un método principal. Se hace que sea parte integrante de la función padre y solo esta puede acceder.

El siguiente código muestra el aislamiento de un proceso dentro de la función local `ObtenerNombreFichero`.

```
public void GenerarFactura(string idFactura)
{
    string fichero = ObtenerNombreFichero(idFactura);

    //Aquí se implementarían los procesos que permiten
    //tratar los datos de la factura e insertarlos en un fichero
    //...
    string ObtenerNombreFichero(string idFac)
    {
        //ejemplo: FA-BU6004295-20170413.pdf
        return $"FA-{idFac}-{DateTime.Now.ToString("yyyyMMdd")}.pdf";
    }
}
```

Como cualquier bloque de código, una función local puede acceder a los miembros de sus padres. En este caso es posible simplificar la función `ObtenerNombreFichero` eliminando su parámetro `idFac` y utilizando directamente el valor del parámetro `idFactura` de su función padre.

```
public void GenerarFactura(string  idFactura)
{
    string fichero =  ObtenerNombreFichero();

   //Aquí se implementarían los procesos que permiten
    //tratar los datos de la factura e insertarlos en un fichero
    //...

    string ObtenerNombreFichero()
    {
        //ejemplo: FA-BU6004295-20170413.pdf
        Return $"FA-{idFactura}-{DateTime.Now.ToString("yyyyMMdd")}.pdf";
    }
}
```

Las tuplas

Cuando una función debe devolver varios valores, el uso de parámetros de escritura es una solución muy funcional. Por otro lado, puede ser tedioso debido a la pesadez de escritura que implica. Para evitar esto, la solución más coherente es devolver los diferentes valores en forma de conjunto. En C#, estos conjuntos de datos se llaman objetos: esto se explicará en detalle en el siguiente capítulo. La séptima versión del lenguaje integra la creación y el uso de objetos simples que se pueden utilizar para devolver varios valores: las tuplas.

Una tupla se describe bajo la forma de un conjunto de tipos entre paréntesis. Cada uno de los tipos corresponde a un dato que se puede integrar en el conjunto de datos. La siguiente línea permite declarar una tupla que contiene dos valores enteros.

```
(int, int) miVariable;
```

Las tuplas pueden tener más de dos valores, e incluso un número virtual ilimitado de datos. De todos modos no se aconseja utilizarlas para agrupar un gran número de elementos. La siguiente declaración de variables agrupa tres valores enteros y una cadena de carácteres, y representa unas coordenadas espaciales X, Y, Z y un identificador asociado.

```
(int, int, int, string) coordenadas;
```

La asignación de un valor a una variable de tipo tupla tiene una nomenclatura parecida a su declaración. Los valores se pasan como una lista entre paréntesis.

```
(int, int, int, string) coordenadas = (0,0,0, « Origen ») ;
```

También es posible asignar los diferentes valores individualmente accediendo a las variables cuyo nombre se gestiona secuencialmente en la forma `Item<index>`. En el caso de la variable `coordenadas`, se puede acceder a los tres valores numéricos a través de los campos `Item1`, `Item2` e `Item3`. El nombre asociado al punto es la variable `Item4`.

```
(int, int, int, string) coordenadas;
coordenadas.Item1 = 0 ;
coordenadas.Item2 = 0 ;
coordenadas.Item3 = 0 ;
coordenadas.Item4 = « Origen » ;
```

En este punto, el principal defecto que se le puede reconocer a las tuplas es el del nombre de sus miembros. Así, es muy fácil confundir las diferentes variables. Para resolver este problema, basta con añadir nombres explícitos en la declaración de la tupla.

```
(int X, int Y, int Z, string Nombre) coordenadas;
coordenadas.X = 0 ;
coordenadas.Y = 0  ;
coordenadas.Z = 0 ;
coordenadas.Nombre = « Origen »  ;
```

Volvamos ahora a la problemática inicial. Las funciones pueden devolver varios valores asociados en forma de tupla. El ejemplo utilizado aquí es una función que devuelve a la vez la suma y la diferencia de dos números enteros.

```
public (int suma, int diferencia) SumarYRestar(int num1,
int num2)
{
    var suma = num1 + num2 ;
    var diferencia = num1 - num2 ;

    return (suma, diferencia) ;
}



var resultado = SumarYRestar (5, 2) ;
Console.WriteLine(« Suma : « + resultado.suma)  ;
Console.WriteLine(« Diferencia : « + resultado.diferencia) ;
```

Las tuplas permiten simplificar el retorno de valores múltiples, pero este mecanismo puede ocasionar una transferencia de complejidad del método llamado al método que invoca. De hecho, es necesario utilizar una notación con punto para acceder a los valores de una tupla.

Para remediar este problema vinculado a la construcción de las tuplas, se ha implementado la lógica inversa en C# con el nombre de deconstrucción. Allí donde la creación de la tupla permite agrupar elementos, la deconstrucción permite separar la tupla en diferentes valores.

Para ello, se utiliza una sintaxis similar para la declaración de diferentes variables que recibirán los valores de una tupla.

```
(var suma, var diferencia) = SumarYRestar(5, 2);
Console.WriteLine(« Suma: « + suma);
Console.WriteLine(« Diferencia: « + diferencia);
```

Incluso es posible "factorizar" la palabra clave `var` para simplificar todavía más la lectura y escritura del código.

```
var (suma, diferencia) = SumarYRestar (5, 2);
```

Además, la deconstrucción se puede utilizar para dar valor a variables que ya existieran. En este caso, la palabra clave `var` no es necesaria.

```
int suma;
int diferencia;
(suma, diferencia) = SumarYRestar (5, 2);
Console.WriteLine(« Suma: « + suma);
Console.WriteLine(« Diferencia: « + diferencia);
```

Los atributos

Los atributos son elementos de código que permiten proveer información relativa a otros elementos de una aplicación.

Se almacenan en los **metadatos** de un ensamblado en tiempo de compilación y pueden leerse una vez finaliza la compilación. Pueden describir un ensamblado, proveer información útil para la depuración o dar alguna pista sobre el uso de un elemento de código. Es posible definir atributos sobre una gran cantidad de elementos del código. Se listan en la enumeración `System.AttributeTargets` e incluyen, en particular, los ensamblados, las clases y los métodos.

Un atributo se define situando en el código el nombre de su tipo entre los símbolos [y]. Por convención, los tipos de atributos son un nombre que termina en `Attribute`, aunque es posible acortar este nombre cuando se utilizan omitiendo el sufijo. Las dos siguientes declaraciones de clase producen exactamente el mismo efecto.

```
[ObsoleteAttribute]
public class  Client
{
}
```

```
[Obsolete]
public class Client
{
}
```

La reflexión

El framework .NET incluye un juego de clases y métodos que permite leer los metadatos de los ensamblados, de los que forman parte los atributos. Se encuentran principalmente en el espacio de nombres `System.Reflection`. Las posibilidades que ofrecen estos tipos permiten implementar escenarios avanzados, como el uso de clases desconocidas por la aplicación (principio de funcionamiento de los plug-ins) o la creación de sistemas de mapeo objeto-relacional (transformación automática de datos desde una base de datos en objetos .NET y a la inversa).

Principios de la programación orientada a objetos

La noción de objetos es omnipresente cuando se desarrolla con C#. En primer lugar veremos lo que representa esta noción y a continuación estudiaremos cómo ponerla en práctica.

La programación procedural, tal y como se utiliza en lenguajes como C o Pascal, define los programas como flujos de datos que sufren transformaciones, en el hilo de la ejecución, por procedimientos y funciones. No existe ningún vínculo robusto entre los datos y las acciones.

La programación orientada a objetos (POO) introduce la noción de conjunto coherente de datos y de acciones, trasponiendo al mundo del desarrollo conceptos comunes e intuitivos propios del mundo que nos rodea.

En efecto, utilizamos, de manera cotidiana objetos con todas las propiedades y acciones que tienen asociadas. Pueden también interactuar entre sí o estar compuestos unos por otros, lo que permite formar sistemas complejos.

Existe una infinidad de analogías para materializar este concepto, pero escogeremos como ejemplo para esta introducción un automóvil, que es lo suficientemente sencillo y lo suficientemente complejo para ilustrar perfectamente las nociones asociadas a la POO.

Un coche tiene propiedades que le son propias, como por ejemplo su marca o color, así como acciones asociadas, como el hecho de arrancar, frenar o encender las luces y puede producir eventos como una luz en el cuadro de mandos, en caso de ocurrir algún problema mecánico.

La existencia de estos elementos en el código C# se traduce en la escritura de clases. Una clase es un modelo que define las propiedades y las acciones de cada uno de los objetos que se crearán a partir de él. La etapa de creación de un objeto se denomina instanciación. En el caso de nuestra analogía, podemos considerar que una clase es como el plano de un coche que se proporciona a la fábrica. La instanciación se corresponde, en sí misma, con la etapa de fabricación. Como ocurre en una fábrica, es posible construir una cantidad infinita de coches a partir del mismo modelo.

La ejecución de la acción para encender las luces sobre un coche implica el uso de ciertos elementos que están ocultos al usuario: se encuentran debajo del capó y no deberían utilizarse por separado. Esto nos va a permitir introducir uno de los tres pilares esenciales en la POO: la encapsulación.

Encapsulación

Cada objeto expone los datos y funciones que permiten a los demás objetos interactuar con él y solo estos. Todos los datos propios del funcionamiento del objeto, que son inútiles desde un punto de vista de las interacciones con el exterior, permanecen ocultos. El objeto se comporta, de este modo, como una caja negra que expone solamente ciertos datos y funcionalidades cuyo uso no podrá corromper el estado del objeto.

En el caso de nuestra analogía, podríamos decir que un objeto Coche expone el dato *Nivel de carburante* de lectura y de escritura, el dato *Estado de las luces* de solo lectura y la funcionalidad *Encender las luces*, entre otras.

Confirmamos que el estado de las luces no puede modificarse salvo ejecutando la acción *Encender las luces*, lo que permite garantizar que el estado de las luces se corresponde con el estado del interruptor correspondiente. Por otro lado, el nivel de carburante puede modificarse si rellenamos o vaciamos el tanque, lo cual no puede corromper el estado interno del vehículo, y el estado *Averiado*, aunque desagradable, es perfectamente lógico y previsible.

Para definir el segundo fundamento de la POO es posible presentar el automóvil como un concepto genérico a partir del que se conciben todos los modelos. Todos los coches tienen el mismo funcionamiento lógico, a pesar de los destalles de diseño que los hacen diferentes: todos se sustentan en cuatro ruedas, poseen un volante, un motor, así como una carrocería en la que se incrustan todos los elementos. De este modo, en el contexto de la POO, podemos asegurar que cualquier vehículo hereda las propiedades y funcionalidades de este automóvil genérico.

Herencia

La herencia define una relación de tipo "es un" entre dos clases. Esta relación significa que una clase expone las mismas funcionalidades y datos que otra clase, agregando potencialmente sus propias funcionalidades. Generalmente, se utiliza la herencia para implementar la noción de especialización.

En nuestro caso será posible decir que un coche de tipo 4x4 "es un" automóvil, como sería posible decir que un coche de tracción delantera "es un" automóvil. De este modo, es posible definir la clase `Automóvil` como la clase base de la que heredan las clases `Coche4x4` y `CocheTracciónDelantera`. Las dos clases hijas pueden aprovechar las funcionalidades implementadas en `Autómovil` e implementar, a su vez, funcionalidades específicas al modo de funcionamiento que utilizan.

La herencia es una noción profundamente integrada en .NET y por tanto en C#. En efecto, todos los tipos heredan, directamente o no, de la clase `System.Object` que es, en consecuencia, la clase madre de todas las demás clases y que no hereda de ninguna otra clase. Esta relación de herencia es implícita: si una clase no hereda explícitamente de otra, entonces hereda de `System.Object`.

Cuando una clase es la clase hija en una relación de herencia, se dice que deriva de su clase madre.

Polimorfismo

El último concepto fundamental de la POO está en relación directa con la herencia. El polimorfismo es, en efecto, la capacidad de un objeto para ser visto según distintas formas. Pero estas formas no son aleatorias, sino que dependen directamente de la jerarquía de herencia de la que forma parte el objeto. Cada clase puede verse en su propia forma, pero también en la de su clase madre, así como según todas las clases "ancestras" de su clase madre. Cada objeto tiene, por tanto, como mínimo dos formas posibles: la definida por su propia clase y la definida según el tipo `Object`.

En el caso de la herencia que acabamos de estudiar es posible, por tanto, afirmar que un objeto `CocheTracciónDelantera` también puede accederse como un objeto de tipo `Automóvil` y como un objeto de tipo `System.Object`. Este comportamiento permite crear funcionalidades relativas a los objetos de tipo `CocheTracciónDelantera` y `Coche4x4` haciendo referencia a objetos de tipo `Automóvil`.

Clases y estructuras

Para abordar el uso de objetos, corazón de la POO, veremos cómo crearlos, declarando la clase o la estructura que servirá como modelo hasta su instanciación. Veremos, también, cómo agregar propiedades y funcionalidades a nuestros objetos.

1. Clases

La mayoría de tipos definidos en el framework .NET son clases, de modo que resulta muy importante comprender cómo manipularlas. En efecto, cualquier aplicación escrita en C# contendrá, como mínimo, una clase escrita por el desarrollador y utilizará probablemente varias decenas o centenares propias del framework .NET.

a. Declaración

Una clase se define y se utiliza mediante su nombre. Es preciso respetar ciertas reglas en su nomenclatura, de modo que puede resultar imposible de compilar si no se respetan. Las posibilidades de uso de la clase son relativas a las definidas por el modificador de acceso asociado.

Nombre de una clase

El nombre de una clase es válido solamente si se respetan las siguientes reglas:

- Contiene únicamente caracteres alfanuméricos o el carácter _.
- No empieza por una cifra.

Por convención, es habitual nombrar las clases respetando el estilo "UpperCamelCase" (también llamado PascalCase), es decir, la primer letra de cada palabra que compone el nombre de la clase se escribe en mayúsculas, mientras que el resto se escribe en minúsculas. Todas las clases del framework .NET respetan esta convención.

`MiClase`, `MuroDeLadrillos` o `PortaEquipajes` son nombres que respetan la regla "UpperCamelCase".

`pantallaPlana` es un nombre válido pero que no respeta esta convención.

Sintaxis

Una clase se declara utilizando la palabra clave `class` seguida de un nombre y de un bloque de código delimitado por los caracteres { y }.

La sintaxis general de declaración de una clase es la siguiente:

```
<modificador de acceso> [partial] class <nombre de la clase> :
[<Clase base>, <Interfaz1>, <Interfaz2>, ...]
{
}
```

Las nociones de clase base y de interfaz se estudiarán en este capítulo dentro de las secciones La herencia y Las interfaces.

Las clases son de tipo referencia, lo que significa que las variables cuyo tipo es una clase tienen como valor por defecto `null`. Este comportamiento es importante y es conveniente conocerlo, pues implica que es necesario instanciar un objeto para inicializar la variable. La noción de instanciación se estudia un poco más adelante en este capítulo (consulte la sección Uso de clases y estructuras).

Modificadores de acceso

Los modificadores de acceso son palabras clave que definen la visibilidad de la clase para el resto del código. Los distintos modificadores de acceso disponibles para las clases son los siguientes, enumerados del más al menos restrictivo:

- `private`: este modificador es válido únicamente en el caso de clases anidadas, y en ese caso la clase es visible únicamente por el tipo que la contiene (consulte el siguiente ejemplo).
- `protected`: este modificador es válido únicamente en el caso de clases anidadas, y en ese caso la clase es visible únicamente por el tipo que la contiene y sus tipos derivados (consulte el siguiente ejemplo).
- `internal`: la clase es visible por todas las cases definidas en el ensamblado.
- `protected internal`: la clase es visible por todas las clases definidas en el ensamblado y por una clase exterior al ensamblado únicamente si hereda de ella.
- `public`: la clase es visible por cualquier otro tipo, sin importar a qué ensamblado pertenezca.


Cuando no se indica ningún modificador de acceso en la declaración de una clase, el compilador marca la clase como `internal`.

Ejemplo de clase anidada con visibilidad `private`:

```
public class Coche
{
    private class Transmisión
    {
    }
}
```

En el caso anterior, la clase `Transmisión` será visible únicamente por `Coche`. Si el modificador de acceso se remplaza por `protected`, `Transmisión` sería una clase accesible por `Coche` y por sus clases hijas únicamente.

Variables de instancia

Las clases representan conjuntos coherentes de datos y acciones. Por tanto, es posible declarar variables en las clases. Se las denomina variables miembro, o miembros de instancia. Se declaran de la misma manera que las variables de una función, salvo por el hecho de que es posible asignarles visibilidad.

Por defecto, si no se indica ningún modificador de acceso para una variable, el compilador considera que la variable está marcada como `private`.

```
public class Coche
{
    //Declaración de una variable miembro pública
```

```
    public int kilometraje;
}
```

Para utilizar una variable de instancia, basta con prefijar el nombre de la variable con el nombre del objeto sobre el que desea interactuar y el operador punto (`.`). En este caso, para modificar el kilometraje de nuestro vehículo, podríamos escribir:

```
//miFerrari es, evidentemente, de tipo Coche
miFerrari.kilometraje = 42;
```

b. Constructor y destructor

Como ocurre con los objetos físicos, los objetos que se utilizan en C# tienen un ciclo de vida definido. Este ciclo comienza con la instanciación del objeto y finaliza con su destrucción y limpieza por parte del Garbage Collector. Para ejecutar instrucciones durante cada una de estas dos operaciones, existen métodos específicos: el constructor y el destructor.

Constructor

Un constructor es un método especial que permite instanciar un objeto. Su nombre es idéntico al de la clase en la que se ha definido, y no tiene ningún tipo de retorno.

Todas las clases poseen, al menos, un constructor (a excepción de las clases abstractas, que veremos en la sección La herencia), esté o no definido en el código. Si no se declara explícitamente ningún constructor, el compilador crea un constructor vacío que no recibe ningún parámetro, llamado constructor por defecto.

A menudo, resulta muy útil definir un constructor para inicializar las variables miembro de la clase.

La sintaxis que permite escribir un constructor es la siguiente:

```
<modificador de acceso> <Nombre de la clase>([<parámetro1>,
<parámetro2>...])
{
}
```

Es posible sobrecargar los constructores, lo que permite inicializar el objeto de varias maneras en función del contexto de uso.

```
public class Empleado
{
    private int edad;
    private int salario

    public Empleado()
    {
    }

    public Empleado(int edadEmpleado)
    {
```

```
            edad = edadEmpleado;
    }

    public Empleado(int edadEmpleado, int salarioEmpleado)
    {
            edad = edadEmpleado;
            salario = salarioEmpleado;
    }
}
```

Es posible declarar uno o varios constructores con los modificadores de acceso `private` o `protected`. Esto puede resultar muy útil cuando se necesita controlar completamente las instanciaciones de una clase, como en el caso de la implementación de un patrón de diseño de tipo Singleton.

El patrón de diseño Singleton permite asegurar que el número máximo de instancias de una clase es en todo momento 1.

C# 6 introdujo un acceso directo sintáctico para la escritura de algunos elementos de código sencillos. Un constructor cuyo cuerpo contiene una sola instrucción se puede definir con el operador => asociado a esta instrucción. La sintaxis utilizada es la siguiente:

```
<modificador de acceso> <Nombre de la clase>([parámetros]) => <expresión>
```

Utilizando esta sintaxis, el constructor que acepta un solo parámetro se puede escribir así:

```
    public Empleado(int edadEmpleado)
        => edad = edadEmpleado;
```

Destructor (o finalizador)

La destrucción de un objeto es una operación automática en una aplicación .NET. El Garbage Collector (GC) supervisa la totalidad de instancias creadas a lo largo de la ejecución de la aplicación y marca como huérfano cada objeto que no se utiliza. Cuando el sistema necesita más memoria, el GC destruye aquellos objetos que considera que es posible destruir invocando a sus destructores, si los tienen.

Un destructor es, por tanto, otro método particular cuya firma está impuesta. Su nombre es el mismo que el de la clase, precedido por el carácter ~. Además, no puede recibir ningún parámetro y, como el constructor, no tiene ningún tipo de retorno.

Este método, si está implementado, debe ejecutar el código necesario para liberar eventuales recursos no gestionados que no se liberarían de otra forma, como archivos o conexiones a bases de datos.

El destructor de nuestra clase `Coche` puede escribirse de la siguiente manera:

```
~Coche()
{
    //Incluir aquí el código que permite limpiar
    //los recursos utilizados por el objeto
```

```
}
```

El acceso directo sintáctico incorporado por C# 6 también es válido para los destructores.

```
~Coche() => <instrucción única>;
```

Es imposible prever cuándo el GC destruirá un objeto particular, pero es posible, en cualquier caso, pedirle que realice una recogida de objetos inutilizados y liberar memoria. Esta operación es costosa, en términos de recursos, para obtener un beneficio que puede no ser demasiado evidente (desde algunos kilobytes a varias decenas de megabytes en función del contexto). Para ejecutar esta recogida, basta con ejecutar el método estático `Collect` de la clase `GC`:

```
GC.Collect();
```


Para conocer el número de bytes asignados a una aplicación en memoria, es posible utilizar el siguiente código:

```
long memoriaUtilizada = GC.GetTotalMemory(false));
```

La impredecibilidad del GC hace que los desarrolladores aíslen el código de limpieza en un método que se invoca directamente cuando el objeto ya no se utiliza.

Estos métodos se llaman, por lo general, `Dispose` y forman parte de la implementación del patrón de diseño **Disposable**.

```
public void Dispose()
{
    //Código de liberación de recursos
}
```

La llamada a este método puede omitirse por parte del desarrollador que utiliza la clase, conviene invocarlo desde el destructor para estar seguros de que el recurso será liberado.

Puede plantearse un problema: el método `Dispose` puede invocarse una vez explícitamente en el código y otra vez implícitamente mediante el destructor. Es necesario, por tanto, establecer un mecanismo que permita ejecutar varias veces el método `Dispose` sin causar un error.

```
public class Coche
{
    private bool estáLimpio = false;

    public void Dispose()
    {
            if (!estáLimpio)
            {
                  //Código de limpieza
                  estáLimpio = true;
```

```
            }
    }

    ~Coche()
    {
            Dispose();
    }
}
```

Otra solución sería indicar al GC que ya no es necesario invocar al destructor del objeto una vez se haya limpiado.

```
public void Dispose()
{
    //Código de liberación de recursos

    GC.SuppressFinalize(this);
}
```

c. Clases parciales

La versión 2 del lenguaje C# (aparecida con .NET 2.0) introduce la noción de clases parciales para gestionar mejor la generación automática de código. En efecto, las interfaces gráficas Windows Forms se crean por completo mediante código C#. La presencia de código generado y de código escrito por los desarrolladores en el mismo archivo era una fuente recurrente de problemas. Con las clases parciales es posible dividir una clase en varias porciones que pueden estar repartidas en varios archivos.

Es posible, de este modo, generar el código de una interfaz gráfica Windows Forms de clases de acceso a los datos dejando la posibilidad de extender este código sin crear nuevas clases o provocar situaciones de conflicto entre el código generado y el código del desarrollador.

Es posible declarar una clase parcial de la misma manera que una clase clásica, pero la palabra clase `class` debe estar prefijada por la palabra clave `partial`. Todas las partes de una clase parcial deben declararse con el mismo nombre en el mismo ensamblado y en el mismo espacio de nombres. Si no se respetan estas condiciones, el compilador considerará las distintas partes como clases diferentes, lo cual, evidentemente, no coincide con el resultado esperado.

```
public partial class Empleado
{
    public int salario;
}

public partial class Empleado
{
    public int edad;
}
```

En tiempo de compilación, ambas declaraciones se fusionarán. De este modo, será posible escribir:

```
//La variable empleado es de tipo Empleado
```

```
empleado.edad = 19;
empleado.salario = empleado.salario * 1.1;
```

2. Estructuras

Las estructuras son, como las clases, conjuntos coherentes de datos y funcionalidades. Son similares a las clases, pero tienen como principal diferencia que son **tipos valor**.

Esta diferencia implica comportamientos diferentes, en particular el hecho de que un objeto de tipo estructura no puede ser `null` y tiene, obligatoriamente, un valor. Tras la declaración de una variable de tipo estructura, cada uno de los miembros del objeto está inicializado a su valor por defecto.

Una estructura se declara prácticamente igual que una clase, pero utilizando la palabra clave `struct`.

```
<modificador de acceso> [partial] struct <nombre de la estructura>
{
}
```

Implícitamente, todas las estructuras heredan de la clase `System.Value Type`, la cual hereda de la clase `System.Object`, pero no pueden heredar de otras clases o estructuras. No pueden tener destructor ni constructores sin parámetros. En cambio, no es obligatorio definir un constructor en una estructura.

```
public struct Coche
{
    public int kilometraje;
}
```

Para inicializar nuestro coche podemos escribir:

```
Coche miFerrari;
miFerrari.kilometraje = 42;
```

Cabe destacar que de la misma manera que con las clases, es posible definir estructuras parciales. Éstas se utilizan con poca frecuencia, pues los objetos de tipo estructura son, la mayoría de veces, contenedores ligeros destinados a agrupar diversas variables con alguna relación entre sí.

3. Creación de un método

Un método es una función o un procedimiento definido en una clase o una estructura. Define una funcionalidad asociada al objeto y permite manipular variables del objeto para modificar su estado.

a. Creación

Para definir un método, basta con escribir un procedimiento o una función en el interior de una definición de tipo.

```
public class Coche
```

```
{
    private bool estáArrancado;

    public void Arrancar()
    {
            estáArrancado = true;
            Console.WriteLine("RUN RUN");
    }
}
```

La sintaxis simplificada vista anteriormente también es válida para los métodos que solo tienen una instrucción.

```
public void Arrancar() => estáArrancado = true;
```

En esta sintaxis simplificada hay que tener en cuenta una sutileza. Cuando una función (que devuelve un valor) se reescribe de esta manera, la palabra clave `return` no debe aparecer.

```
   public bool EstáArrancado()
   {
           return estáArrancado;
   }
```

se convierte en:

```
   public bool EstáArrancado () => estáArrancado;
```

Una vez definido, un método puede, como una variable, invocarse sobre un objeto gracias al operador punto (**.**), siempre que el modificador de acceso del método lo autorice.

```
miFerrari.Arrancar();
```

Varios métodos pueden tener el mismo nombre, pero deben diferenciarse por sus argumentos. Se dice, en tal caso, que los métodos están sobrecargados.

La distinción entre las diferentes sobrecargas no se realiza en base al nombre de los argumentos. Los aspectos importantes son:

- El número de argumentos,
- El tipo de los argumentos,
- El orden de los argumentos.

```
public void EnviarCorreo(string nombreDestinatario, Dirección
direcciónDestinatario)
{
}
```

```
public void EnviarCorreo(Dirección direcciónDestinatario, string
nombreDestinatario)
{
}

public void EnviarCorreo(string nombreDestinatario, Dirección
direcciónDestinatario, decimal precioSello)
{
}

public void EnviarCorreo(string nombreDestinatario, Dirección
direcciónDestinatario, string direcciónRemitente)
{
}
```

Escribir estos cuatro métodos en la misma clase es perfectamente válido, incluso aunque el segundo tiene un interés muy limitado pues recibe los mismos parámetros que el primero. Cada llamada al método se vinculará con la declaración correcta en tiempo de compilación, en función de los argumentos que se pasen en la llamada.

b. Métodos parciales

La versión 3 de C# (aparecida con .NET 3.5) incluye métodos parciales por el mismo motivo que existen las clases parciales desde su aparición con C# 2. La generación automática de código **LINQ to SQL** se lleva a cabo gracias a clases parciales que permiten al desarrollador agregar funcionalidades al código generado. La aparición del concepto de los métodos parciales permite, además, agregar procesamientos en el código generado, sin modificarlo.

El principio de los métodos parciales es relativamente simple: una parte del método se define con la palabra clave `partial` pero no posee bloque de código asociado. Esto permite al compilador saber que el método es susceptible de tener una implementación en otra parte de la clase.

Los métodos parciales deben estar definidos en clases parciales. En caso contrario, el compilador genera un error.

Los métodos parciales no pueden ser funciones. En efecto, el código que invoca a esta función dependería de ellas para poder funcionar correctamente, lo que no se corresponde con la lógica de extensión de funcionalidad descrita antes.

Por el mismo motivo no pueden tener modificadores de acceso, y su visibilidad se define directamente como `private` por parte del compilador.

```
partial class Coche
{
    partial void EnArranque();

    void Arrancar()
    {
           EnArranque();
    }
}
```

La segunda parte del método puede definirse en el mismo archivo o en otro archivo diferente, pero siempre dentro de la misma clase parcial. Debe tener la misma firma que la primera parte.

```
partial class Coche
{
    partial void EnArranque()
    {
            Console.WriteLine("RUN RUN");
    }
}
```

De este modo, tras la llamada al método `Arrancar`, ¡el motor ruge!

Si la segunda parte del método no está presente, entonces el método parcial no tiene implementación. En este caso, el compilador suprime la definición del método parcial así como las llamadas a dicho método.

c. Métodos de extensión

Los métodos de extensión permiten agregar funcionalidades a las clases o a las estructuras existentes sin modificar su código. Este tipo de método está adaptado a la extensión de clases del framework .NET tales como la clase `String`. En efecto, estas clases no pueden modificarse y ocurre con frecuencia que se desea modificar su comportamiento agregando alguna funcionalidad.

Antes de C# 3, la funcionalidad se definía en una clase de utilidades, eventualmente estática y que se invocaba de manera convencional.

```
bool esPalíndromo = objetoUtilidad.EsPalíndromo(miCadena);
```

Como los métodos utilidades descritos anteriormente, los métodos de extensión se escriben en el exterior de la clase extendida pero pueden utilizarse de la misma manera que los métodos de la clase extendida.

Un método de extensión debe respetar una convención de escritura estricta:

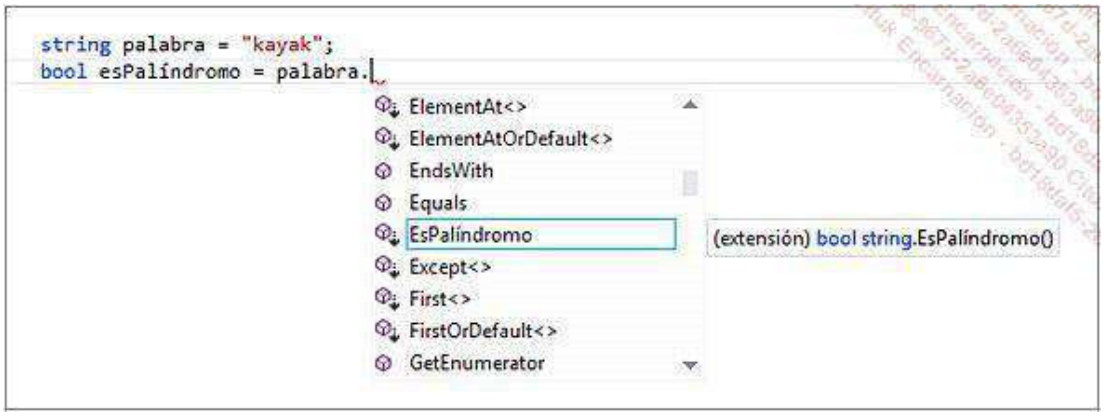
- Debe definirse en una clase marcada como `static`.
- Debe estar marcada como `static`.
- Su primer parámetro (obligatorio) debe ser de tipo extendido. Debe estar precedido por la palabra clave `this`.

En el siguiente ejemplo agregamos un método a la clase `string` que permite comprobar si una cadena es palíndroma, es decir, si puede leerse indistintamente de izquierda a derecha o de derecha a izquierda.

```
static class Extensions
{
    public static bool EsPalíndromo(this string cadenaAComprobar)
    {
            if (cadenaAComprobar == null)
                    return false;

            string cadenaInversa = "";
            for (int i = cadenaAComprobar.Length - 1; i >= 0; i--)
            {
```

```
                cadenaInversa = cadenaInversa + cadenaAComprobar [i];
            }

            return cadenaInversa.Equals(cadenaAComprobar);
        }
}
```

Si utilizamos, a continuación, una variable de tipo `string`, el método `EsPalíndromo` está disponible y lo sugiere IntelliSense.



Los métodos de extensión se recuperan en la lista con un icono diferente a los métodos clásicos así como con la precisión "(extension)" en el texto informativo asociado.

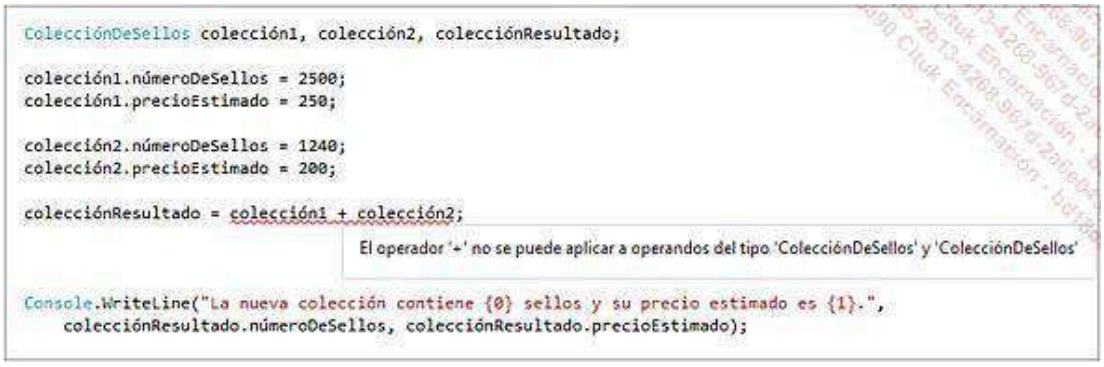
d. Métodos operadores

Los métodos operadores permiten sobrecargar un operador estándar del lenguaje para utilizarlo sobre tipos definidos por el desarrollador.

En el caso de dos colecciones de sellos, sobrecargar el operador + permitiría fusionar las colecciones.

Tomemos dos objetos de tipo `ColecciónDeSellos` definidos como se muestra a continuación y probemos a sumarlos:

```
struct ColecciónDeSellos
{
    public int númeroDeSellos;
    public int precioEstimado;
}
```



El compilador no sabe cómo sumar ambos elementos y nos lo indica mostrando un mensaje de error bloqueante para la compilación. Si sobrecargamos el operador + para nuestra estructura:

```
public static ColecciónDeSellos operator +(ColecciónDeSellos
colección1, ColecciónDeSellos colección2)
{
    ColecciónDeSellos resultado;
    resultado.numeroDeSellos = colección1.numeroDeSellos +
colección2.numeroDeSellos;
    resultado.precioEstimado = colección1.precioEstimado +
colección2.precioEstimado;

    return resultado;
}
```

Una vez agregado este código, el error de compilación desaparece y la ejecución del código muestra el siguiente resultado:

```
La nueva colección contiene 3740 sellos y su precio
estimado es de 450€
```

4. Creación de propiedades

Las clases pueden contener datos públicos con la forma de variables miembro, pero para respetar el concepto de encapsulación, es preferible exponer propiedades en lugar de variables. En efecto, las propiedades permiten ejecutar código con cada acceso de lectura o de escritura. Esto permite validar los datos y, de este modo, mantener el objeto en un estado coherente.

a. Lectura y escritura

Una propiedad se asemeja a una declaración de función pero contiene dos bloques de código identificados por las palabras clave `get` y `set`. El bloque `get` se ejecuta con la lectura de la propiedad mientras que el bloque `set` se ejecuta cuando se accede a la propiedad en escritura.

Una propiedad tiene el siguiente aspecto:

```
private int kilometraje;
```

```
public int Kilometraje
{
    get
    {
        return kilometraje;
    }
    set
    {
        if (value >  kilometraje)
        {
            kilometraje =  value;
        }
        else
        {
            Console.WriteLine("El kilometraje de un coche no
puede disminuir");
        }
    }
}
```

Para respetar el principio de encapsulación la variable `kilometraje` se marca como privada con `private`.

Podríamos destacar la palabra clave `value` que encontramos en el bloque `set`. Representa al valor asignado a la propiedad. Se realiza una comprobación en el bloque `set` para validar el valor que se pasa como argumento a la propiedad.

Los bloques `get` y `set` de las propiedades se pueden utilizar con una sintaxis simplificada cuando tienen una sola instrucción. Se puede ver que en el bloque `get` se omite la palabra clave `return:` está presente de modo implícito ya que el bloque debe devolver un valor.

```
private int kilometraje;

public int Kilometraje
{
    get => kilometraje;
    set => kilometraje = value ;
}
```

b. Solo lectura

Puede resultar interesante, para la coherencia de un objeto, exponer una propiedad únicamente en solo lectura. En nuestro caso, podríamos decir que el kilometraje de un vehículo no debería modificarlo más que el funcionamiento interno del mismo. Proteger la variable de instancia `kilometraje` exponiéndola únicamente en solo lectura parece, en este caso, una excelente solución.

Para ello, basta con declarar la propiedad sin bloque `set`. Sin este bloque, cualquier intento de modificar el valor de la propiedad produce un error de compilación. ¡Nuestro kilometraje está a salvo!

```
private int kilometraje;

public int Kilometraje
```

```
{
    get => kilometraje;
    }
}
```

Una segunda propiedad que proporcione el valor del kilometraje seguido del símbolo "km" podría escribirse de la siguiente manera.

```
public string KilometrajeString => string.Format("{0} km", kilometraje);
```

c. Solo escritura

En casos muy asilados, podría resultar útil definir una propiedad de solo escritura. Por ejemplo, podríamos querer definir una propiedad `Contraseña` en una clase `Usuario`. Es evidente que no queremos que este campo pueda leerse. Una propiedad de solo escritura parece, por tanto, una solución adecuada para implementar este comportamiento.

Una propiedad de solo escritura se implementa omitiendo el bloque `get`.

```
private string contraseña;

public string Contraseña
{
    set => contraseña = value;
    }
}
```

d. Propiedades automáticas

Cuando no se asocia ninguna lógica particular a los métodos de acceso `get` y `set` de una propiedad, puede resultar interesante utilizar las propiedades automáticas, aparecidas con C# 3 y .NET 3.5. Estos elementos se declaran mediante un atajo sintáctico para la definición de las propiedades.

Una propiedad automática se declara como una propiedad clásica, pero no se facilita el cuerpo de los bloques `get` y `set`. Agreguemos una propiedad `Color` a nuestro `Coche`:

```
public string Color { get; set; }
```

Como esta propiedad no accede explícitamente a una variable miembro, no es necesario declarar una para almacenar el valor asignado. Entre bastidores, el compilador crea una y genera el cuerpo de los bloques `get` y `set`. A continuación se muestra una representación en C# del código IL generado por el compilador.

```
private string color ;
public string Color
{
    get => return color;
    set => color = value;
```

```
}
```

Uno u otro de los bloques `get` y `set` puede estar marcado por un modificador para proteger el acceso a la propiedad (y solo exponer públicamente la propiedad en solo lectura, por ejemplo).

```
public string Color { get; private set; }
```

El modificador utilizado por un bloque debe ser más restrictivo que el modificador aplicado a la propiedad. Especificar un modificador de acceso a ambos bloques produce un error de compilación, incluso aunque los bloques tengan el mismo modificador de acceso.

e. Inicialización de propiedades automáticas

Hasta la quinta versión del lenguaje C#, no era posible inicializar una propiedad automática tras su declaración: era obligatorio hacerlo mediante un bloque de código, como un método, o en el cuerpo de otra propiedad. Desde C# 6 ya es posible realizar esta acción. La sección de código siguiente declara una propiedad automática y le asigna el valor "Verde".

```
public string Color { get; set; } = "Verde";
```

El valor utilizado para la inicialización puede, también, estar devuelto por una expresión, como una llamada a una función o a un cálculo.

```
public string Color { get; set; } = EncontrarNombreColorRGB(0, 255, 0);
```

f. Propiedades automáticas de solo lectura

La posibilidad de inicializar una propiedad en el momento de su declaración aporta a C# la capacidad de crear propiedades automáticas de solo lectura. Sin inicialización, una propiedad automática de solo lectura no tendría, en efecto, ninguna utilidad.

Este tipo de propiedad se declara de la misma forma que una propiedad automática con inicialización. La única diferencia es la eliminación de la palabra clave `set` para obtener una propiedad de solo lectura. La propiedad `Color` definida más arriba puede declararse de solo lectura de la siguiente manera:

```
public string Color { get; } = "Verde";
```

A continuación, se muestra una representación C# del código IL generado por el compilador.

```
private readonly string color;
public string Color
{
    get => color;
}
```

El código IL del constructor de la clase que contiene la propiedad se ve, también, modificado para incluir una instrucción que inicializa la variable miembro `color`.

La palabra clave `readonly` utilizada en la declaración de la variable miembro generada indica que esta no es modificable más que en el constructor de la clase que la contiene.

g. Propiedades indexadas

Las propiedades indexadas, también llamadas indexadores o propiedades por defecto, permiten realizar un acceso de tipo array a un conjunto de elementos.

Estas propiedades reciben uno o varios parámetros que se pasan entre corchetes. No tienen nombre y una clase puede poseer varias propiedades indexadas que deben diferenciarse por sus parámetros.

Una propiedad indexada se declara utilizando la siguiente sintaxis:

```
<modificador de acceso> <tipo> this[parámetro1, [parámetro2,
parámetro3...]]
{
    //Accesores get y/o set
}
```

Una implementación (totalmente inútil) de una propiedad indexada podría ser la siguiente:

```
public class Coche
{
    public string this[int numeroRueda]
    {
          get => return "Rueda n°" + numeroRueda;
    }
}
```

Esta propiedad indexada puede utilizarse de la siguiente manera:

```
//coche es de tipo Coche
Console.WriteLine(coche[2]);
```

Lo que devuelve: `Rueda n° 2`

5. Miembros estáticos

Los miembros estáticos (o miembros compartidos) de una clase son variables, propiedades o métodos que no son específicos de una instancia de la clase, sino de la propia clase. Un miembro estático existe en un único ejemplar. Acceder a este miembro a través de un objeto no tendría sentido, por lo que se accede mediante el nombre del tipo asociado. Intentar utilizarlo desde una instancia produce un error de compilación.

Un miembro estático no puede tampoco acceder a un miembro de instancia de su clase, salvo si se le provee el objeto del que se quiere utilizar el miembro estático.

Es posible definir un miembro compartido utilizando la palabra clave `static` tras el modificador de acceso.

Aquí, consideramos que el número de ruedas es una característica idéntica para todos los coches. Es, por tanto, posible agregar una propiedad estática que defina este número en la clase `Coche`:

```
  public static int NumeroDeRuedas get => 4;
```

Para utilizar esta propiedad, escribiremos:

```
int numero = Coche.NumeroDeRuedas;
```

Es igual para todos los tipos de miembros que pueda tener una clase o estructura.

La palabra clave `static` no puede combinarse con la palabra clave `const`, las constantes son particulares a los miembros estáticos.

6. Uso de clases y estructuras

El caso de uso general de una clase o de una estructura pasa por la construcción de un objeto. Una vez creado, es posible leer o escribir datos en el objeto, así como ejecutar sus métodos asociados.

a. Instanciación

La instanciación de un objeto a partir de una clase se realiza mediante la palabra clave `new`:

```
//Coche es una clase

//Declaración de la variable. Como Coche es una clase,
//auto vale null
Coche auto;

//Instanciación del objeto. Ya no es null
auto = new Coche();
```

Para las estructuras las cosas funcionan de una manera algo diferente. Como una estructura es de tipo valor, una variable estructura no puede valer `null` en el momento de su declaración, y cada uno de sus miembros tiene como valor el valor por defecto de su tipo.

```
//Coche es una estructura

//Declaración de la variable
Coche auto;
```

Si se define un constructor para el tipo estructura, entonces es posible inicializar nuestro objeto de la misma manera que se instancia un objeto a partir de una clase.

```
Coche auto;
//El constructor de Coche recibe como parámetro: int kilometraje
auto = new Coche(42);
```

b. Inicialización

La inicialización de un objeto es un proceso en el que los elementos son la construcción de una instancia y la asignación de valores a sus propiedades. De manera clásica, se inicializan las propiedades de un objeto de la siguiente manera:

```
Coche auto = new Coche();
auto.Color = "Rojo";
auto.Kilometraje = 12500;
auto.Precio = 20000;
```

Para simplificar este proceso, que puede resultar algo tedioso, C# 3 introduce los inicializadores que permiten crear nuestro objeto e inicializar completa o parcialmente sus propiedades públicas en una única instrucción.

```
Coche auto = new Coche { Color = "Rojo", Kilometraje =
12500 };
auto.Precio = 20000;
```

c. Tipos anónimos

Durante el desarrollo de una aplicación, a menudo resulta útil crear pequeños objetos que se utilizarán en una o dos funciones a lo sumo. Estos objetos tienen un tiempo de vida muy corto y no tienen ninguna responsabilidad visible de cara al funcionamiento de la aplicación.

Crear clases como plantilla para estos objetos es una restricción que no aporta, realmente, ningún valor a la aplicación.

Desde C# 3 ya no es necesario crear tipos específicos para estos objetos. En efecto, los tipos anónimos realizan exactamente el rol que acabamos de describir.

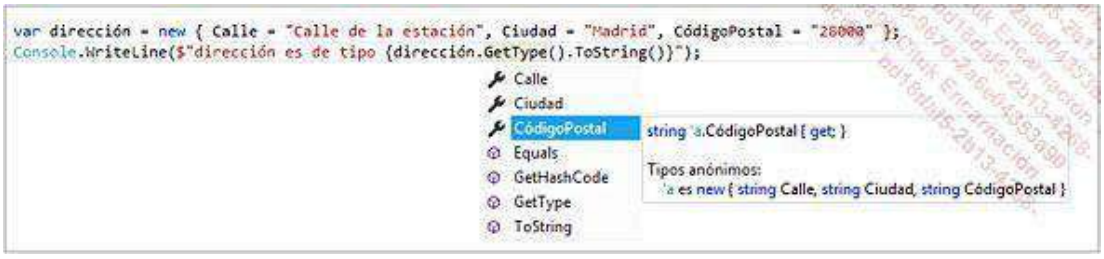
Es posible crear un objeto de tipo anónimo utilizando prácticamente la misma sintaxis que hemos visto para los inicializadores. Basta con no especificar el nombre del tipo tras la palabra clave `new` y declarar cada una de nuestras propiedades asignándoles un nombre y un valor.

```
new { Color = "Rojo", Kilometraje = 12500 }
```

Por el contrario, se plantea un problema... ¿Cómo declarar la variable que va a contener al objeto?

Efectivamente, es el caso de uso típico donde la **inferencia de tipo** y la palabra clave `var` de C# muestran su valor. En efecto, el compilador C# es capaz de generar un tipo a partir de las propiedades que hemos especificado en la creación de nuestro objeto. Este tipo desconocido en tiempo de desarrollo cobrará vida en tiempo de compilación y la variable declarada mediante la palabra clave `var` estará vinculada en ese momento a este nuevo tipo.

Para comprobarlo, hagamos una prueba:



Antes incluso de la compilación, Visual Studio nos muestra que nuestro tipo se reconoce como anónimo y que posee, efectivamente, tres propiedades de tipo `string`. Posee, también, cuatro métodos que puede reconocer: se trata de los métodos provistos por la clase `System.Object`. En efecto, como cualquier clase, nuestro tipo anónimo hereda también de la clase `Object`.

Estos métodos son los únicos que podrá tener nuestro objeto, pues es imposible definir un método en un objeto anónimo.

El resultado de la ejecución del código es el siguiente:

```
dirección es de tipo
<>f_AnonymousType0`3[System.String,System.String,System.String]
```

Podemos constatar que el compilador ha generado, efectivamente, un tipo (con un nombre algo grotesco) y ha vinculado la variable `dirección` a este tipo. Por el contrario, instanciar una variable de tipo anónimo no quiere decir, necesariamente, que el compilador generará una clase para esta variable. Buscará en primer lugar un tipo ya generado y que se corresponda con la variable que deseamos crear, y creará uno si no encuentra ninguna correspondencia.

Para comprender qué criterios internos utiliza el compilador, creemos algunas variables y comparemos el resultado con nuestro resultado anterior:

```
//2 variables declaradas exactamente de la misma manera
var dirección = new { Calle = "Calle de la estación", Ciudad = "Madrid", CódigoPostal = "28000" };
var direcciónIdéntica = new { Calle = "Calle de la estación", Ciudad = "Madrid", CódigoPostal = "28000" };
//4 propiedades en lugar de 3
var direcciónNúmero = new { Número = 15, Calle = "Calle de la estación", Ciudad = "Madrid", CódigoPostal = "28000" };
//Una propiedad con un nombre diferente
var direcciónNombre = new { Calle = "Calle de la estación", Ciudad = "Madrid", CP = "28000" };
//CódigoPostal y Ciudad se han intercambiado
var direcciónOrden = new { Calle = "Calle de la estación", CódigoPostal = "28000", Ciudad = "Madrid" };
//CódigoPostal es, ahora, un valor entero
var direcciónTipo = new { Calle = "Calle de la estación", Ciudad = "Madrid", CódigoPostal = 28000 };

Console.WriteLine($"dirección es de tipo {dirección.GetType()}");
Console.WriteLine($"direcciónIdéntica es de tipo {direcciónIdéntica.GetType()}");
Console.WriteLine($"direcciónNúmero es de tipo {direcciónNúmero.GetType()}");
Console.WriteLine($"direcciónNombre es de tipo {direcciónNombre.GetType()}");
Console.WriteLine($"direcciónOrden es de tipo {direcciónOrden.GetType()}");
Console.WriteLine($"direcciónTipo es de tipo {direcciónTipo.GetType()}");
```

El resultado es el siguiente:

```
dirección es de tipo
<>f_AnonymousType0`3[System.String,System.String,System.String]

direcciónIdéntica es de tipo
<>f_AnonymousType0`3[System.String,System.String,System.String]

direcciónNúmero es de tipo
<>f_AnonymousType1`4[System.Int32,System.String,System.String,
System.String]

direcciónNombre es de tipo
<>f_AnonymousType2`3[System.String,System.String,System.String]

direcciónOrden es de tipo
<>f_AnonymousType3`3[System.String,System.String,System.String]

direcciónTipo es de tipo
<>f_AnonymousType0`3[System.String,System.String,System.Int32]
```

Las dos primeras variables son del mismo tipo, pero todas las demás tienen un tipo diferente. Deducimos que los criterios de elección son los siguientes:

- El nombre de las propiedades,
- El número de propiedades,
- El orden de definición de las propiedades,
- El tipo de las propiedades.

Cabe destacar que es posible comparar dos instancias de la misma clase anónima utilizando el método `Equals` (heredado de la clase `Object`). El compilador lo sobrecarga en el tipo generado y permite indicar que las variables `dirección` y `direcciónIdéntica` son iguales:

```
Console.WriteLine(dirección.Equals(direcciónIdéntica));
```

El método `Equals` devolverá false si alguna de las condiciones no se respeta:

- Los dos objetos comparados deben ser del mismo tipo.
- Deben declararse en el mismo ensamblado.
- Los valores de cada propiedad de los objetos deben ser idénticos.

Los espacios de nombres

El código de una aplicación puede contener facilmente varias decenas, centenares o millares de tipos diferentes. Para organizar el proyecto es posible crear una estructura jerárquica de carpetas que agrupan los tipos por utilidad o en base a su contexto de uso. Además de esta organización física, es posible crear una organización lógica mediante el concepto de espacio de nombres (**namespace**, en inglés).

1. Nomenclatura

Un espacio de nombres está compuesto por varios identificadores separados por el operador `.`, cada uno de estos identificadores forma parte de un contenedor lógico. Veamos algunos ejemplos de espacios de nombres:

```
System
System.Windows
System.Data.SqlClient
```

Estos tres espacios de nombres forman parte del framework .NET. `System` contiene los tipos básicos de.NET, como los tipos primitivos, `System.Windows` es el contenedor lógico de tipos básicos que permiten crear aplicaciones de ventanas. Por último, `System.Data.SqlClient` contiene los tipos del bloque ADO.NET específicos de la base de datos SQL Server.

Además de ayudar a la estructuración, el uso de espacios de nombres permite tener varios tipos cuyo nombre es idéntico: para evitar cualquier ambigüedad entre estos tipos es posible utilizar el nombre plenamente cualificado del tipo que se desea utilizar. Este nombre completo es la concatenación del espacio de nombres, del operador punto (`.`) y del nombre del tipo.

El framework .NET incluye varias clases llamadas `DataGrid`. Estas clases se encuentran en espacios de nombres diferentes y es posible hacer referencia a cada uno de estos tipos sin ambigüedad de la siguiente manera:

```
System.Windows.Controls.DataGrid dataGridWpf;

System.Windows.Forms.DataGrid dataGridWindowsForms;
```

Para incluir un tipo en un espacio de nombres basta con declararlo en el interior de un bloque `namespace`.

```
namespace MiAplicación.Rss
{
    public class LectorFlujoRss
    {
    //...
    }
}
```

2. using

A menudo, no será necesario utilizar este nombre completo explícitamente. Es posible, en efecto, declarar en el código que queremos utilizar las clases de uno o varios espacios de nombres particulares mediante la palabra clave

using.

En este contexto, esta palabra clave no tiene nada que ver con el uso que se ha detallado en el capítulo Las bases del lenguaje, en la sección Otras estructuras.

De este modo, si deseamos utilizar las clases del espacio de nombres `System.Windows.Forms`, podemos escribir el código siguiente:

```
using System.Windows.Forms;

public class MiClase
{
    private void CrearDataGrid()
    {
          DataGrid miDataGrid;
          // ...

          //Es posible, también, utilizar el tipo  DataGrid
          //del espacio de nombres  System.Windows.Controls
          //especificando el nombre completo
          System.Windows.Controls.DataGrid  miDataGridWpf;
    }
}
```

Desde C#6, la palabra clave `using` puede utilizarse también para simplificar el uso de elementos estáticos. Los miembros estáticos siempre se utilizan, en efecto, a través del nombre del tipo al que pertenecen, lo que puede producir numerosas repeticiones. La instrucción siguiente utiliza dos miembros de la clase `System.Math` para calcular el coseno de π.

```
using System;

public class MiClase
{
    private void EjecutarCálculo()
    {
          int cosPi = Math.Cos(Math.PI);
          ...
    }
}
```

El uso de la combinación de palabras clave `using static` seguido del nombre de un tipo estático permite librarse de tener que utilizar el nombre de este tipo en el archivo de código. El ejemplo anterior podría, de este modo, reescribirse de la siguiente manera:

```
using System;
using static Math;

public class MiClase
{
    private void EjecutarCálculo()
```

```
    {
        int cosPi = Cos(PI);
        ...
    }
}
```

Preste atención: el uso de `using static` puede llevar a una mala legibilidad del código, e incluso producir ciertos conflictos que impidan la compilación. Podría resultar difícil, en efecto, tanto para el desarrollador como para el compilador determinar a qué clase pertenece una función utilizada en este contexto.

La herencia

La herencia permite representar e implementar una relación de especialización entre una **clase de base** y una **clase derivada**. Permite, por tanto, transmitir las características y el comportamiento del tipo de base hacia el tipo derivado, así como su modificación.

1. Implementación

Para declarar una relación de herencia entre dos clases es necesario modificar la declaración de la clase que se quiere definir como clase derivada. Esta modificación consiste en agregar el símbolo "**:**" seguido del nombre de la clase de base tras la declaración del tipo:

```
public class ClaseDeBase
{
    public int Identificador;

    public void MostrarIdentificador()
       => Console.WriteLine(Identificador)
}

public class ClaseDerivada : ClaseDeBase
{
}
```

A diferencia de otros lenguajes, como C++, C# no autoriza a derivar un tipo a partir de varias clases de base.

Una vez implementada esta relación de herencia es perfectamente posible escribir el siguiente código:

```
ClaseDerivada miObjeto = new ClaseDerivada();
miObjeto.Identificador = 42;
miObjeto.MostrarIdentificador();
//Muestra 42 en la consola
```

En efecto, `ClaseDerivada` puede acceder a los miembros `public`, `internal` y `protected` de su clase de base, incluidas la variable `Identificador` y el método `MostrarIdentificador`.

2. Las palabras clave this y base

La palabra clave `this` devuelve la instancia en curso de la clase en la que se utiliza. Permite, por ejemplo, pasar una referencia del objeto en curso a otro objeto.

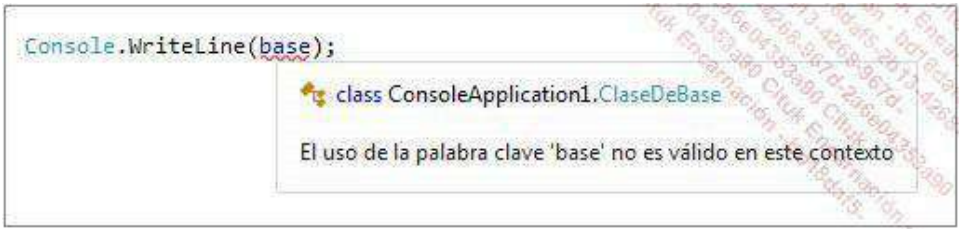
```
namespace ThisYBase
{
    public class Coche
    {
          public decimal Longitud;
```

```
        //Constructor de la clase Coche
        public Coche()
        {
            //Las 2 líneas siguientes tienen,  exactamente,
            //el mismo rol pues devuelven el objeto en  curso

            Longitud = 4.2;
            this.Longitud = 4.2;

            //La siguiente línea escribe en la consola
            //el resultado de la llamada  this.ToString()
            //("ThisYBase.Coche", en este caso)

            Console.WriteLine(this);
        }
    }
}
```

La palabra clave `base` permite hacer referencia al objeto en curso, pero bajo la forma de su clase de base. Esta palabra no se utiliza sola: es obligatorio utilizarla para hacer referencia a un miembro de la clase de base. No es posible, por tanto, escribir el siguiente código:

```
Console.WriteLine(base);
```

El compilador C# nos devuelve, en efecto, un error bastante explícito:



En cambio, es posible escribir lo siguiente:

```
Console.WriteLine(base.ToString());
```

Podríamos esperar que el resultado mostrado por la consola sea el tipo de la clase de base (`System.Object`), pero no es el caso, dado que obtenemos "ThisYBase.Coche". La explicación es muy sencilla, la detallamos a continuación.

En primer lugar, el código del método `ToString()` de la clase `System.Object` de la que hereda la clase `Coche` implícitamente es el siguiente:

```
public virtual string ToString()
      => return GetType().ToString();
```

La palabra clave `virtual` y su rol se abordan en la siguiente sección: Sobrecarga y ocultación.

El método `GetType` invocado por este método devuelve el tipo instanciado, es decir, el tipo "real" del objeto y no el tipo bajo la forma en la que vemos al objeto. El resultado de la llamada a `base.ToString()` es, por tanto, perfectamente lógico.

3. Sobrecarga y ocultación

En una relación de herencia puede ser necesario modificar un comportamiento para adaptarlo a las características de una clase privada. Existen dos maneras de hacer esto en C#: la sobrecarga y la ocultación. Estas dos técnicas se corresponden con la reescritura de una función en una clase hija, pero se las utiliza en contextos distintos.

a. Sobrecarga de métodos

La sobrecarga de un método se aplica cuando el método se define en la clase madre con la palabra clave `virtual`. Esta palabra clave indica que el método se ha pensado para que se sobrecargue.

Consideremos una clase de base `Persona` que defina un método marcado como `virtual`.

```
public class Persona
{
    public virtual void Conducir()
    {
            //Código para conducir respetando el código de la vía
    }
}
```

El método `Conducir` está marcado como `virtual` pues ciertos tipos de personas pueden, potencialmente, ejecutar la misma acción de distinta manera. Una persona cualquiera debe conducir respetando el código de la vía, un piloto conduce para realizar su trayecto lo más rápidamente posible y un conductor de autobús debe frenar con frecuencia para dejar que suban o bajen los pasajeros.

En una clase `Piloto` que herede de `Persona` sería preciso, por tanto, sobrecargar el método `Conducir` para tener en cuenta esta diferencia. Para ello, se reescribe el método con un código específico en la clase `Piloto` y se marca con la palabra clave `override`.

```
public class Piloto : Persona
{
    public override void Conducir()
    {
            //Código para conducir lo más rápido posible
    }
}
```

El método `override` debe tener la misma firma y la misma visibilidad que el método de la clase de base.

b. Ocultación de métodos

La ocultación de un método es similar a la sobrecarga, pero sus efectos son diferentes (se estudian en la sección

siguiente: Diferencias entre sobrecarga y ocultación). Permite reescribir un método en una clase derivada, pero en la ocultación se utiliza la palabra clave `new` en lugar de la palabra clave `override`.

```
public class Piloto : Persona
{
    public new void Conducir()
    {
            //Código para conducir lo más rápido posible
    }
}
```

No es necesario que el método esté marcado como `virtual` en la clase de base para poder ocultarlo.

c. Diferencias entre sobrecarga y ocultación

Si bien ambas nociones pueden parecer idénticas, son ligeramente diferentes en su efecto.

La clase `Persona` define dos métodos, `Conducir_Override` y `Conducir_New`. La clase `Piloto` hereda de `Persona` y sobrecarga el primer método, mientras que oculta el segundo.

```
public class Persona
{
    public virtual void Conducir_Override()
    => Console.WriteLine("Conduzco respetando el código de
circulación");

    public void Conducir_New()
    => Console.WriteLine("Conduzco respetando el código de
circulación");

}

public class Piloto : Persona
{
    public override void Conducir_Override()
       => Console.WriteLine("Conduzco lo más rápido posible
para ganar");

    public new void Conducir_New()
       => Console.WriteLine("Conduzco lo más rápido posible
para ganar");
}
```

Sobrecarga

Ejecute el siguiente código:

```
Persona persona = new Persona();
persona.Conducir_Override();
```

```
Piloto piloto = new Piloto();
piloto.Conducir_Override();

Persona personaPiloto = new Piloto();
personaPiloto.Conducir_Override();
```

El resultado de esta ejecución es el siguiente:

```
Conduzco respetando el código de circulación
Conduzco lo más rápido posible para ganar
Conduzco lo más rápido posible para  ganar
```

Ocultación

Ejecute, ahora, el siguiente código:

```
Persona persona = new Persona();
persona.Conducir_New();

Piloto piloto = new Piloto();
piloto.Conducir_New();

Persona personaPiloto = new Piloto();
personaPiloto.Conducir_New();
```

El resultado de esta ejecución es el siguiente:

```
Conduzco respetando el código de circulación
Conduzco lo más rápido posible para ganar
Conduzco respetando el código de  circulación
```

Comprobamos que existe una diferencia en la tercera operación. La clave está en la siguiente instrucción:

```
Persona personaPiloto = new Piloto();
```

El método `Persona.Conducir_Override` está marcado como `virtual`. La presencia de la palabra clave desencadena el siguiente proceso:

- En tiempo de ejecución, el CLR inspecciona el tipo concreto de la variable `personaPiloto`. Aquí, el tipo es `Piloto`.
- Busca el método sobrecargado `Conducir_Override` en el tipo concreto `Piloto`.

En este momento, existen dos posibilidades:

- El método se encuentra y se ejecuta.

- El método no se encuentra, en cuyo caso el CLR reconsidera el tipo de la clase madre del tipo concreto y encuentra el método buscado.

Cuando se utiliza el operador `new` sobre un método, se corta el árbol de búsqueda.

Si el método `Persona.Conducir_New` está marcado como `virtual`, se ejecuta la búsqueda empezando por la clase madre del primer tipo que oculta el método. Aquí, la clase `Persona`.

Si el método no está marcado como `virtual` al principio, se ejecuta simplemente el método de la clase manipulada, en este caso `Persona.Conducir_New`.

Estos conceptos pueden parecer complicados de entender a primera vista, pero su manipulación hace que sea una tarea más sencilla. No dude en escribir código con relaciones de herencia para asimilarlo mejor.

4. Imponer o prohibir la herencia

En ciertos casos puede resultar interesante establecer ciertas restricciones sobre una clase para limitar sus condiciones de uso, en particular imponer o prohibir la herencia de esta clase. En C# es posible realizar estas acciones mediante dos palabras clave: `abstract` y `sealed`.

Las clases abstractas

Las clases abstractas son tipos definidos como no instanciables. Solamente pueden utilizarse, por tanto, mediante clases derivadas.

Estos tipos se utilizan para centralizar elementos de código comunes a varios tipos o para proveer funcionalidades básicas que deben completarse en un tipo derivado. Pueden contener todos los elementos de código de una clase "normal": métodos, propiedades, variables miembro, etc.

Las clases abstractas pueden, también, contener métodos abstractos, es decir, cuya implementación no se ha provisto. Es necesario, por tanto, que cada tipo derivado provea su propia implementación del método.

La sintaxis de declaración de una clase abstracta requiere agregar la palabra clave `abstract` delante del nombre de la clase.

```
<modificador de acceso> abstract <nombre de clase>
{
}
```

A continuación se muestra un ejemplo de clase abstracta que posee un método abstracto y de la que hereda una clase:

```
public abstract class ClaseAbstracta
{
    public abstract void  MétodoAbstracto();
}

public class ClaseDerivada :  ClaseAbstracta
{
```

```
    public override void  MétodoAbstracto()
    {
            //Implementación del método
    }
}
```

Las clases selladas

Las clases selladas funcionan de manera inversa a las clases abstractas: son, necesariamente, instanciables (no pueden definirse como `abstract`), y es imposible heredar de ellas. Por este motivo, es imposible definir uno de sus miembros como `abstract` o `virtual`, pues sería imposible proveer una implementación para él en una clase derivada.

La sintaxis de declaración de una clase sellada requiere agregar la palabra clave `sealed` delante del nombre de la clase.

```
<modificador de acceso> sealed <nombre de clase>
{
}
```

5. La conversión de tipo

La conversión de tipo es uno de los aspectos fundamentales de la programación orientada a objetos con C#, pues es una de las claves para la buena implementación del polimorfismo. Es, en efecto, lo que permite visualizar un objeto bajo varias formas.

En función del contexto de uso, la conversión puede ser implícita o explícita. Las conversiones implícitas se realizan automáticamente cuando es necesario.

Pueden realizarse cuando se cumplen dos condiciones:

- Los tipos de origen y de destino son compatibles: ambos tipos están ligados por una relación de herencia o en el tipo de origen se ha definido una operación de conversión implícita hacia el tipo de destino.
- La conversión de un tipo hacia otro no supone pérdida de datos.

Las conversiones de tipo implícitas pueden realizarse, entre otros, de un tipo numérico `short` a un `int`, pues no se produce ninguna pérdida de datos, o de un tipo derivado hacia un tipo de base, pues el objeto del tipo derivado es necesariamente un tipo de base.

```
short valorShort = 12;
int valorInt = valorShort;


//El tipo Piloto hereda de Persona
Piloto piloto = new Piloto();
Persona persona = piloto;
```

En cambio, las siguientes líneas de código producen un error en tiempo de compilación:

```
int valorInt = 12;
short valorShort = valorInt;
```

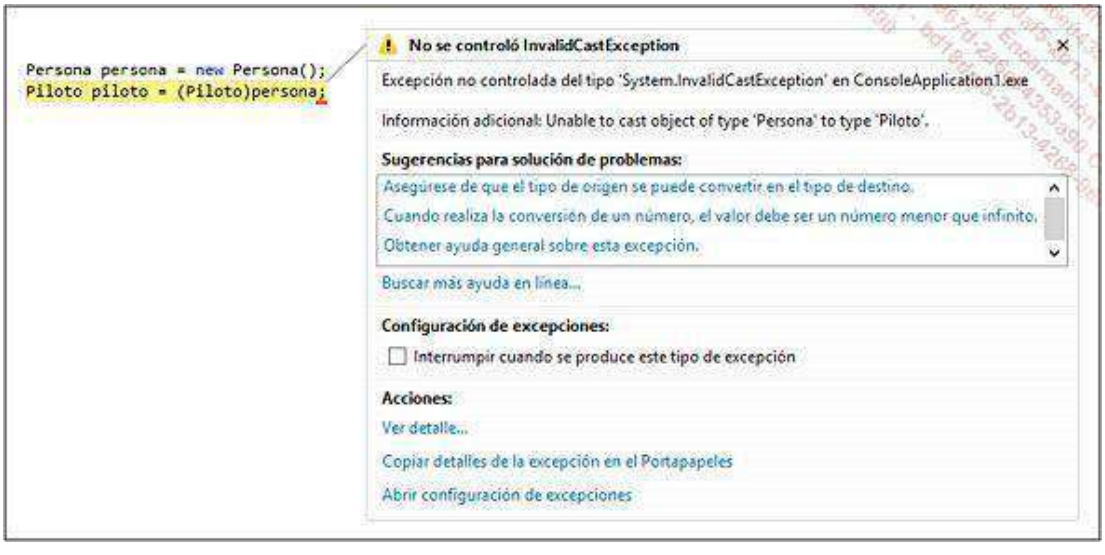
En efecto, incluso aunque se sepa que el valor 42 puede contenerse perfectamente en un `short`, el compilador detecta una posible pérdida de datos en la conversión de tipo `int` (4 bytes) hacia `short` (2 bytes). Es preciso, en este caso, utilizar un **cast (conversión de tipo explícita)** para validar la intención del desarrollador.

El operador de conversión de tipo se representa por unos paréntesis entre los que se indica el tipo de destino. En el caso descrito más arriba, la corrección sería la siguiente:

```
int valorInt = 42;
short valorShort = (short)valorInt;
```

El caso es ligeramente más complicado para convertir la clase `Persona` a la clase `Piloto`. Es preciso, en efecto, que el objeto de tipo `Persona` contenga un objeto de tipo `Piloto` para poder convertirse a este tipo.

El siguiente código genera un error en tiempo de ejecución:



Para eliminar el error, es preciso escribir el siguiente código:

```
Persona persona = new Piloto();
Piloto piloto = (Piloto)persona;
```

Las interfaces

Para manipular distintos tipos de objetos que poseen funcionalidades similares es posible utilizar un contrato que define los datos y los comportamientos comunes a estos tipos. En C# se denomina, a este contrato, una interfaz.

Una interfaz es un tipo que posee únicamente miembros públicos. Estos miembros no poseen implementación: se respetan los tipos a través del contrato y cada miembro aportará su propia implementación.

Las interfaces representan una capa de abstracción, particularmente útil para hacer que una aplicación sea modular, pues permiten utilizar cualquier implementación para un mismo contrato.

1. Creación

Las interfaces se declaran de forma similar a las clases, con las siguientes diferencias:

- Es preciso utilizar la palabra clave `interface` en lugar de `class`.
- Solamente los miembros que sean visibles públicamente deben definirse en la interfaz y no existe ningún otro modificador de acceso autorizado sobre estos miembros.
- Ningún miembro de la interfaz puede tener implementación.

La sintaxis general para crear una interfaz es la siguiente:

```
<modificador de acceso> interface <nombre> [: interfaces de base ]
{

}
```


Por convención, el nombre de las interfaces en C# empieza por una letra I (i mayúscula).

Consideremos el caso de la lectura de archivos de audio. Es posible leer archivos en los formatos MP3, WAV, OGG, MWA, o incluso otros. La lectura de cada uno de estos formatos requiere un decodificador particular, por lo que parece conveniente crear una clase para cada uno de estos tipos de archivo soportados en una aplicación: `LectorAudioMp3`, `LectorAudioOgg`, etc.

Es probable que estas clases sean muy similares y que deban utilizarse en las mismas circunstancias, lo que hace de este conjunto un candidato perfecto para la creación de una interfaz común a los distintos tipos.

Una posible interfaz para estas distintas clases podría ser la siguiente:

```
public  interface ILectorAudio
{
    bool LecturaEnCurso { get; }

    bool EstáEnPausa { get;  }

    void Leer(string rutaArchivo);

    void Pausa();
```

```
    void ReanudarLectura();

    void Stop();
}
```

`LecturaEnCurso` y `EstáEnPausa` son dos propiedades para las que se indica que las clases que implementan la interfaz deben proveer, como mínimo, una lectura pública. Todos los métodos definidos en la interfaz estarán presentes en cualquier case que implemente la interfaz.

2. Uso

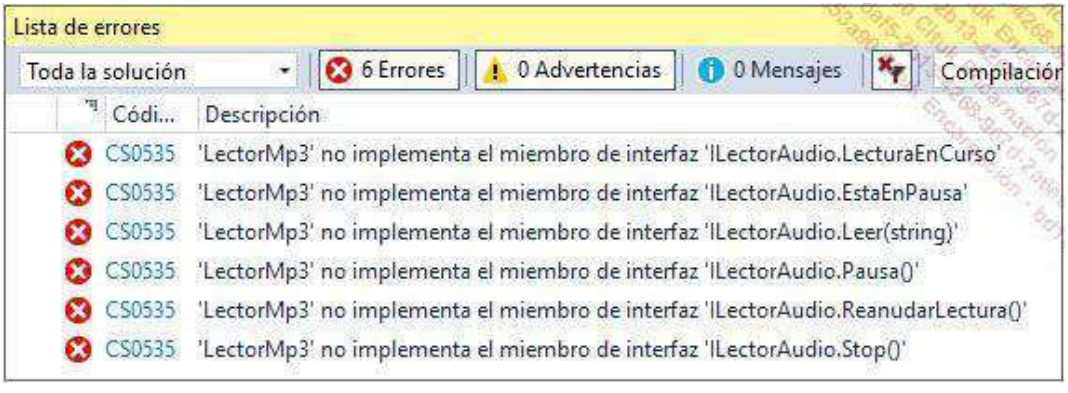
Para utilizar una interfaz, es preciso declarar, en primer lugar, la relación entre la clase y la interfaz. Para ello, se utiliza la misma sintaxis que para la herencia:

```
<modificador de acceso> class <nombre> [:interfaz1, interfaz2...]
{
}
```

La declaración de la clase `LectorMp3` se haría de la siguiente manera:

```
public class LectorMp3 : ILectorAudio
{
}
```

Visual Studio indica que la compilación del código de la interfaz y de la clase, llegados a este punto, produce exactamente seis errores.



El compilador de C# indica claramente que los miembros de la interfaz no se han implementado en la clase `LectorMp3`. Es posible implementarlos de dos maneras: implícita o explícitamente.

a. Implementación implícita

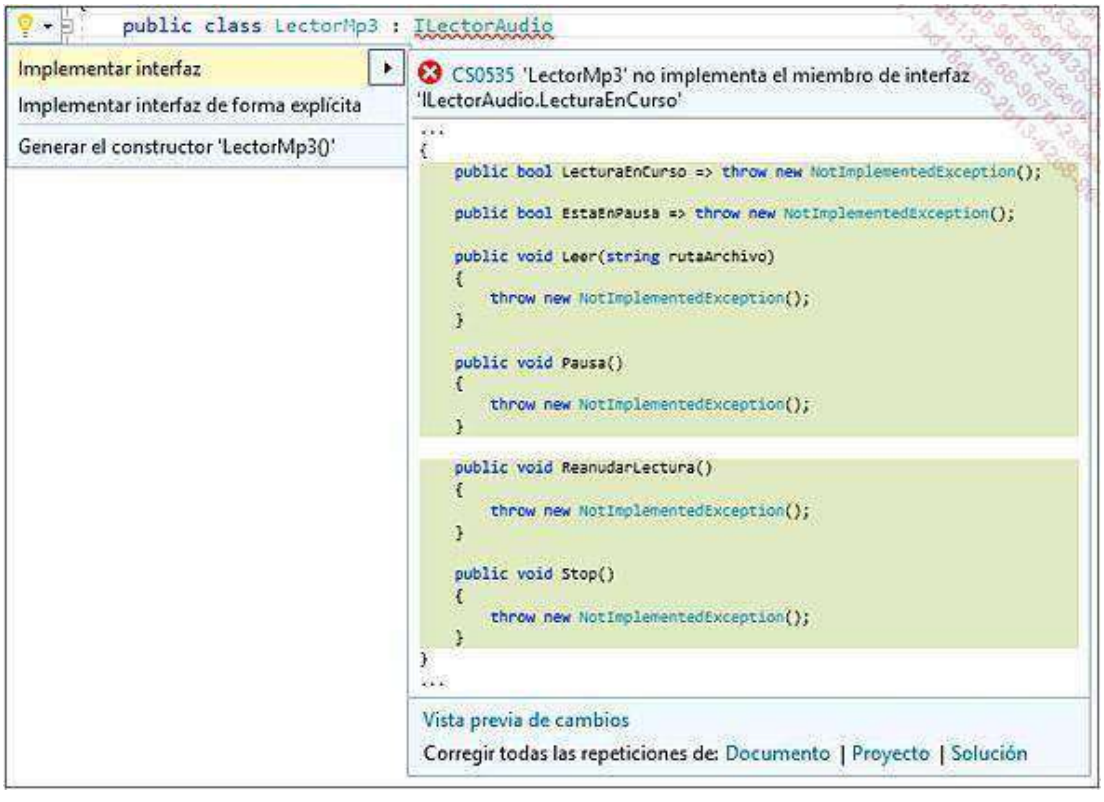
En la mayoría de casos, las interfaces se implementan de manera implícita. Esto quiere decir, simplemente, que cada uno de los miembros definidos en la interfaz existe en las clases que implementan la interfaz.

Cada uno de los miembros debe implementarse con una visibilidad pública y, en el caso de los métodos, con la

misma firma que la definida en la interfaz.

La herramienta de refactorización de Visual Studio puede simplificar esta tarea creando un esqueleto de implementación implícita. Ofrece, también, la posibilidad de obtener la visualización previa de las modificaciones correspondientes a la generación de este esqueleto.

Haciendo clic en el nombre de una interfaz no implementada hace que se visualice una bombilla al inicio de la línea. Al pasar el ratón por encima, esta bombilla se convierte en un botón que permite desplegar un menú contextual destinado a la refactorización del código.



Utilizando esta herramienta se produce el siguiente código:

```
class LectorMp3 : ILectorAudio
{
    public bool LecturaEnCurso => throw new NotImplementedException();

    public bool EstáEnPausa => throw new  NotImplementedException();

    public void Leer(string rutaArchivo)
        => throw new NotImplementedException();

    public void Pausa()
        => throw new NotImplementedException();

    public void ReanudarLectura()
        => throw new NotImplementedException();

    public void Stop()
        => throw new NotImplementedException();
```

```
}
```

Una vez realizada la operación, cada uno de los miembros de la interfaz `ILectorAudio` posee una implementación (no funcional) en la clase `LectorMp3`.

b. Implementación explícita

Puede llegar a darse el caso de que una clase implemente varias interfaces en las que ciertos miembros compartan la misma firma. En este caso, implementar las interfaces implícitamente provoca que se comparta una misma implementación para todas las interfaces afectadas.

Puede, por tanto, ser necesario que la clase incluya una implementación de miembro por interfaz. La implementación explícita es una respuesta a esta problemática particular.

Consideremos las dos interfaces siguientes:

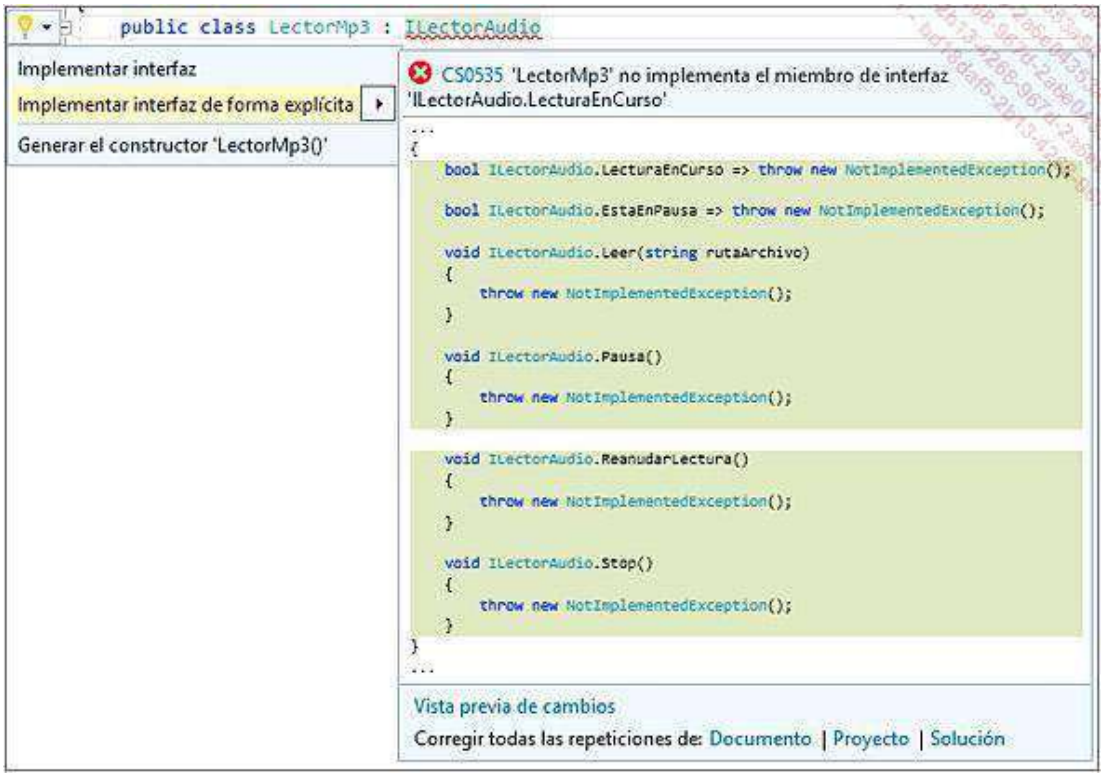
```
public interface ILectorAudio
{
    void Leer(string rutaArchivo);
}
```

```
public interface ILectorVideo
{
    void Leer(string rutaArchivo);
}
```

Para implementar estas dos interfaces en una clase `LectorMultimedia` es necesario que el nombre de cada uno de los miembros específicos esté precedido por el nombre de la interfaz asociada seguido del operador punto.

```
public class LectorMultimedia : ILectorAudio, ILectorVideo
{
    void ILectorAudio.Leer(string rutaArchivo)
    {
            //Arranca la lectura del audio
    }

    void ILectorVideo.Leer(string rutaArchivo)
    {
            //Prepara la representación del flujo de vídeo
            //Arranca la lectura del audio y del vídeo
    }
}
```

Visual Studio también permite generar estas implementaciones explícitas mediante su herramienta de refactorización, con la visualización previa del código generado:


public class LectorMp3 : ILectorAudio
Implementar interfaz
Implementar interfaz de forma explícita
Generar el constructor 'LectorMp3()'
CS0535 'LectorMp3' no implementa el miembro de interfaz 'ILectorAudio.LecturaEnCurso'
...
{
bool ILectorAudio.LecturaEnCurso => throw new NotImplementedException();
bool ILectorAudio.EstaEnPausa => throw new NotImplementedException();
void ILectorAudio.Leer(string rutaArchivo)
{
throw new NotImplementedException();
}
void ILectorAudio.Pausa()
{
throw new NotImplementedException();
}
void ILectorAudio.ReanudarLectura()
{
throw new NotImplementedException();
}
void ILectorAudio.Stop()
{
throw new NotImplementedException();
}
}
...
Vista previa de cambios
Corregir todas las repeticiones de: Documento | Proyecto | Solución

Las enumeraciones

Una enumeración es un tipo de datos que representa a un conjunto finito de valores constantes que pueden utilizarse en las mismas circunstancias.

La declaración de una enumeración se realiza utilizando la siguiente sintaxis:

```
<modificador de acceso> enum <nombre>
{
    <nombre de constante 1> [ = <valor  numérico>],
    <nombre de constante 2> [ = <valor  numérico>],
    ...
}
```

El framework .NET define un número importante de tipos enumerados. Entre ellos encontramos el tipo `System.Windows.Visibility`, que define el estado visual de un control de WPF. Su definición es la siguiente:

```
public enum Visibility
{
    Visible = 0,
    Hidden = 1,
    Collapsed = 2
}
```

Se utilizará un valor de la enumeración escribiendo el nombre de la enumeración seguido del operador punto (`.`) y del nombre de una constante de enumeración.

Para ocultar un control `System.Windows.Grid` podríamos escribir, por ejemplo, el siguiente código:

```
//grid es un objeto de tipo System.Windows.Grid
grid.Visibility = System.Windows.Visibility.Collapsed;
```

Los delegados

Un delegado es un tipo que representa una referencia a un método. Gracias a los delegados es posible especificar que un parámetro de un método debe ser una función que posea una lista de parámetros y un tipo de retorno definido. A continuación, es posible invocar a esta función en el código de nuestro método sin conocerla previamente.

1. Creación

La declaración de un delegado utiliza la palabra clave `delegate` seguida de un procedimiento o función. El nombre especificado en esta firma es el nombre del tipo delegado creado.

```
//Creación de un delegado para una función que recibe
//2 parámetros de tipo double y devuelve un double
public delegate double OperaciónMatemática(double operando1,
double operando2);
```

El código anterior crea un nuevo tipo que puede utilizarse en la aplicación: `OperaciónMatemática`.

2. Uso

El tipo `OperaciónMatemática` puede utilizarlo cualquier variable o cualquier parámetro de un método. Es posible utilizar una variable de este tipo como un método, es decir, es posible pasarle parámetros y recuperar su valor de retorno.

```
private static double
EjecutarOperaciónMatemática(OperaciónMatemática
operaciónARealizar, double operando1, double  operando2)
{
     return operaciónARealizar(operando1, operando2);
}
```

Es preciso, a continuación, invocar al método `EjecutarOperaciónMatemática` pasándole los parámetros convenientes. Para ello, es preciso crear un método correspondiente a la firma del delegado y pasarle el nombre del método como valor del parámetro `operaciónARealizar`.

```
private static double Sumar(double operando1, double
operando2)
    => return operando1 + operando2;
```

```
double resultadoSuma =
EjecutarOperaciónMatemática(Suma, 143, 18);
Console.WriteLine(resultadoSuma);
```

3. Expresiones lambda

Una expresión lambda es una función anónima que puede utilizarse en cualquier lugar donde se espere un valor por parte de un delegado.

Las expresiones lambda utilizan el operador => en su declaración. A la izquierda de este operador encontramos una lista de nombres y parámetros correspondientes a los parámetros que especifica el delegado. Esta lista de parámetros se encuentra entre paréntesis. Existen dos casos particulares para esta lista:

- Cuando la expresión lambda no recibe parámetros es preciso indicar los paréntesis.
- Cuando se recibe un único parámetro, los parámetros son opcionales.

La parte derecha de la expresión es un bloque de instrucciones que puede devolver un valor, o no. Este bloque debe estar definido entre llaves, salvo en el caso de que contenga una única instrucción. En este último caso, las llaves son opcionales.

La sintaxis de declaración de una expresión lambda es la siguiente:

```
(parámetro1, parámetro2, ...) => { [instrucciones que pueden
utilizar los valores parámetro1, parámetro2, ...] }
```

La llamada al método `EjecutarOperaciónMatemática` escrita más arriba para la ejecución de una suma puede reescribirse de la siguiente forma:

```
OperaciónMatemática suma = (op1, op2) => { return op1 + op2; };

double resultadoSuma = EjecutarOperaciónMatemática(suma, 143,
18);
Console.WriteLine(resultadoSuma);
```

Los eventos

Los eventos están en el núcleo del desarrollo de aplicaciones con C#. Permiten basar la lógica de la aplicación sobre una serie de procedimientos y de funciones que se ejecutan cuando alguno de sus componentes solicita la ejecución. Es el caso, por ejemplo, de los componentes gráficos: estos pueden desencadenar eventos cuando el usuario realiza alguna acción como, por ejemplo, la selección de un elemento en una lista desplegable o hacer clic sobre un botón.

1. Declaración y producción

Los eventos de C# están basados en el uso de delegados. La idea general es que cada evento puede recibir uno o varios controladores de eventos con una firma definida por un tipo de delegado.

Los eventos generados por las clases del framework utilizan con frecuencia el tipo de delegado `EventHandler` para definir los controladores de eventos que pueden tener asociados. Este delegado se define de la siguiente manera:

```
public void delegate EventHandler(object sender, EventArgs e);
```

El parámetro `sender` se corresponde con el objeto que ha generado el evento, mientras que el parámetro de tipo `EventArgs`, llamado `e`, se utiliza para proveer información a los métodos que tratan el evento. Si no es necesario pasar ningún valor al controlador de eventos es posible utilizar una instancia de la clase `EventArgs`, pero en caso contrario es conveniente pasar algún objeto de un tipo que herede de la clase `EventArgs`.

Existe también el tipo de delegado genérico `EventHandler<TEventArgs>`, que permite pasar información cuando se produce el evento. Los genéricos se estudian un poco más adelante en este capítulo.

Utilizar el tipo `EventHandler` o su equivalente genérico `EventHandler <TEventArgs>` permite mantener cierta coherencia entre los eventos de la aplicación y los generados por las clases del framework .NET.

La sintaxis general para la declaración de un evento es la siguiente:

```
<modificador de acceso> event <tipo de delegado> <nombre de evento>;
```

La declaración de un evento que se produce al final del cálculo de unas facturas puede escribirse de la siguiente manera:

```
public event EventHandler CálculoFacturasTerminado;
```

Es habitual declarar un evento en un método cuyo único objetivo sea desencadenarlo si existen uno o varios controladores para el evento.

```
protected virtual void DesencadenaCálculoFacturasFinalizado(EventArgs e)
{
    EventHandler handler =  CálculoFacturasTerminado;

    //Si se han definido uno o varios  controladores,
    //handler no será null
```

```
    if (handler != null)
    {
            handler(this, e);
    }
}
```

El evento se desencadena en el lugar donde se desea ejecutar el código de los controladores vinculados a dicho evento, en nuestro caso al final de la ejecución del método `CálculoFacturas` de la clase `Facturación`.

```
public class Facturación
{
    //Declaración del evento
    public event EventHandler CálculoFacturasTerminado;

    public void CálculoFacturas()
    {
        //Código potencialmente largo que realiza el cálculo
        //Aquí, simulamos el cálculo
        //incrementando un contador de 1 a 1000
        for (int i = 0; i < 1000; i++)
        {
            i++;
        }

        //Desencadenamos el evento
        EventArgs argumentos = new EventArgs();
        DesencadenaCálculoFacturasTerminado(argumentos);
    }

    protected virtual void DesencadenaCálculoFacturasTerminado(EventArgs e)
    {
        EventHandler handler = CálculoFacturasTerminado;

        //Si se definen uno o varios controladores,
        //handler no será null
        if (handler != null)
        {
            handler(this, e);
        }
    }
}
```

2. Gestión de los eventos

Antes de gestionar el evento `CálculoFacturasTerminado`, ejecutemos nuestro método de cálculo de facturas:

```
Facturación fact = new Facturación();
fact.CálculoFacturas();
```

Este código funciona sin errores. Aquí, está vinculado a la verificación realizada en el cuerpo del método `DesencadenarCálculoFacturasTerminado`. No se ha asociado ningún controlador al evento `CálculoFacturasTerminado`, de modo que no se desencadena. Si no se realizara esta verificación se produciría una excepción del tipo `NullReferenceException` y se interrumpiría el flujo normal de la aplicación.

Agregar un controlador de eventos

Es posible agregar un controlador de eventos mediante el operador += al que se le pasa el nombre de un método correspondiente al tipo de delegado esperado.

```
<objeto>.<nombre de evento> += <nombre del método controlador>;
```

En el caso del evento `CálculoFacturasTerminado`, el tipo de delegado esperado es `EventHandler`, lo que implica que es preciso definir un método correspondiente a la firma de este delegado.

```
private void controlador_CálculoFacturasTerminado(object sender,
EventArgs e)
    => Console.WriteLine("Ha terminado el cálculo de facturas.");
```

Para que se ejecute el controlador es preciso, a continuación, asociarlo al evento antes de que se produzca:

```
Facturación fact = new Facturación();
fact.CálculoFacturasTerminado += controlador_CálculoFacturasTerminado;
fact.CálculoFacturas();
```

Eliminar un controlador de eventos

Es posible eliminar un controlador de eventos de manera similar a como se agrega, la única diferencia es el operador utilizado. De hecho, se utiliza el operador -= para desasociar un controlador.

```
Facturación fact = new Facturación();
fact.CálculoFacturasTerminado += controlador_CálculoFacturasTerminado;
fact.CálculoFacturas();
fact.CálculoFacturasTerminado -= controlador_CálculoFacturasTerminado;
fact.CálculoFacturas();
```

La ejecución de esta sección de código muestra el mensaje "*Ha terminado el cálculo de facturas*." una única vez, pues el segundo cálculo se realiza tras haber desasociado el controlador de eventos.

Los genéricos

Los genéricos son elementos de un programa capaces de adaptarse de cara a proveer las mismas funcionalidades para distintos tipos de datos.

Se introducen con la aparición del framework .NET 2.0 con el objetivo de proporcionar servicios adaptados a varios tipos de datos, manteniendo un tipado fuerte.

Los frameworks .NET 1.0 y .NET 1.1 proporcionaban la clase `ArrayList` para la gestión de listas dinámicas. Este tipo, muy práctico, tenía el inconveniente de que contenía únicamente objetos de tipo `System.Object`, lo que generaba un gran número de conversiones de tipo que hubiera sido preferible evitar. Los genéricos son la respuesta aportada por Microsoft a este problema: la clase `List<T>` remplaza ventajosamente (en la mayoría de casos) a la clase `ArrayList` y permite definir el tipo de los objetos que contiene.

Esto simplifica el código y lo vuelve más seguro, pues el desarrollador elimina gran parte de los problemas derivados de las potenciales conversiones de tipo.

Los elementos genéricos se reconocen mediante los caracteres < y > presentes en sus nombres. Estos símbolos contienen los nombres de los tipos asociados a la instancia de un elemento genérico.

Los genéricos se estudian aquí a través del desarrollo de un tipo que permite gestionar una lista de elementos. Las listas son colecciones que siguen el principio FIFO (*First In, First Out*): es posible acceder, únicamente, al primer elemento de la colección y cuando se agrega un elemento nuevo, éste se sitúa automáticamente en la última posición de la colección. En el framework .NET existe un tipo similar: `System.Collections.Generic.Queue<T>`.

1. Clases

Las clases genéricas permiten manipular de una manera unificada varios tipos de datos. Estos tipos están designados por uno o varios alias en el nombre de la clase y tras la instanciación de un objeto genérico se asignan los tipos concretos a estos alias.

a. Definición de una clase genérica

Para crear una clase genérica es preciso definir una clase, agregarle los caracteres < y > y definir, entre ellos, uno o varios alias de tipo separados por comas. Estos alias permiten manipular variables fuertemente tipadas sin conocer previamente el tipo concreto.

```
public class ListaFIFO<TElemento>
{
}
```

Por convención, los alias de tipo se prefijan mediante la letra T (de tipo).

La declaración de una clase genérica que tiene la capacidad de gestionar dos tipos de datos simultáneamente podría describirse de la siguiente manera:

```
public class ListaDobleFIFO<TElemento1, TElemento2>
{
}
```

b. Uso de una clase genérica

Para instanciar un tipo genérico es preciso utilizar el operador `new` seguido del nombre del tipo genérico. Los alias de tipo deben remplazarse por nombres de tipos concretos.

```
ListaFIFO<string> lista = new ListaFIFO<string>();
```

Esta instanciación indica que cada elemento marcado por el alias de tipo `TElemento` en la clase `ListaFIFO` se tratará como si fuera de tipo `string`. Este componente es válido únicamente para la instancia definida aquí. Es perfectamente posible definir otra instancia que manipule valores enteros o archivos (`System.IO.File`).

Es posible, también, definir estructuras genéricas. Su declaración es similar a la de las clases genéricas. La diferencia reside, por tanto, en el uso de la palabra clave `struct` en lugar de la palabra clave `class`.

2. Interfaces

Una interfaz genérica permite definir un contrato de uso independiente del tipo de los datos manipulados por las clases que lo implementan.

a. Definición de una interfaz genérica

La definición de una interfaz genérica es similar a la definición de una interfaz normal. La diferencia reside en el hecho de que es necesario asociar uno o varios tipos parámetro tras el nombre de la interfaz, como ocurre en la declaración de una clase genérica.

```
public interface IListaFIFO<TElemento>
{
}
```

La convención de escritura de C# indica que el nombre de las interfaces, sean genéricas o no, comience por I (i mayúscula).

La definición de los miembros de la interfaz puede hacer referencia a cada uno de los tipos parámetro de la interfaz, sin obligación de hacerlo. Agregando dos firmas de métodos a la interfaz para agregar y eliminar un elemento, la definición del tipo `IListaFIFO<TElemento>` es la siguiente:

```
public interface IListaFIFO<TElemento>
{
    void AgregarElemento(TElemento elementoAAgregar);

    TElement EliminarElemento();
}
```

Estas firmas utilizan, ambas, el tipo parámetro `TElemento`. Una clase que implemente esta interfaz precisando que el tipo parámetro es `string` (`IListaFIFO<string>`) debería implementar estos métodos de la

siguiente manera:

```
public void AgregarElemento(string elementoAAgregar)
{
    //implementación concreta del método
}

public string EliminarElemento()
{
    //implementación concreta del método
}
```

b. Uso de una interfaz genérica

Las interfaces genéricas se utilizan de la misma manera que las interfaces normales, pero es preciso, evidentemente, especificar cuáles son los tipos parámetro.

```
class ListaEnteros : IListaFIFO<int>
{
    public void AgregarElemento(int elementoAAgregar)
    {
            //implementación concreta del método
    }

    public int EliminarElemento()
    {
            //implementación concreta del método
    }
}
```

En el caso de la clase `ListaFIFO<TElemento>`, es conveniente que el tipo parámetro de la interfaz sea el mismo que el utilizado en la instanciación de un objeto. Es posible, de hecho, escribir la declaración de la clase siguiente para alcanzar este resultado.

```
public class ListaFIFO<TElemento> : IListaFIFO<TElemento>
```

La implementación completa (y no optimizada) de la clase llegados a este punto puede ser la siguiente:

```
public class ListaFIFO<TElemento> : IListaFIFO<TElemento>
{
    private TElemento[] arrayInterno;

    public ListaFIFO()
=> arrayInterno = new TElemento[0];

    public void AgregarElemento(TElemento elementoAAgregar)
    {
        TElemento[] arrayTemporal = new
```

```
TElemento[arrayInterno.Length + 1];

        arrayInterno.CopyTo(arrayTemporal, 0);
        arrayTemporal[arrayInterno.Length] = elementoAAgregar;

        arrayInterno = arrayTemporal;
    }

    public TElemento EliminarElemento()
    {
        TElemento resultado = arrayInterno[0];

        TElemento[] arrayTemporal = new
TElemento[arrayInterno.Length - 1];
        for (int i = 1; i < arrayInterno.Length; i++)
        {
            arrayTemporal[i - 1] = arrayInterno[i];
        }

        arrayInterno = arrayTemporal;

        return resultado;
    }
}
```

3. Restricciones

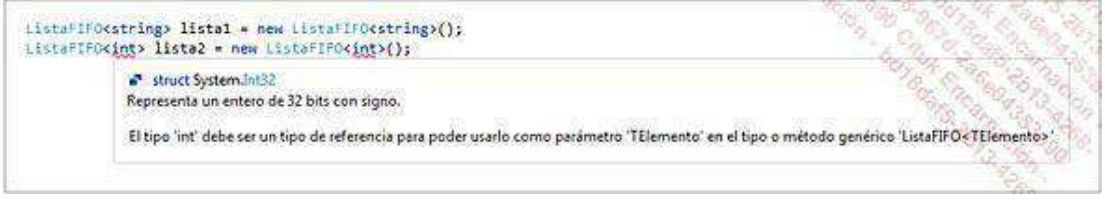
Los elementos genéricos tienen la capacidad de adaptarse a la manipulación de varios tipos de datos. Puede ser adecuado, no obstante, restringir el juego de tipos parámetro que pueden utilizarse para satisfacer ciertas necesidades lógicas o técnicas. El compilador es capaz de verificar que los tipos parámetro de los elementos genéricos responden bien a las restricciones fijadas. Cuando no se respeta alguna de las restricciones se produce un error de compilación.

Es posible fijar cinco tipos de restricciones sobre los tipos parámetro y es posible combinarlas.

where TParametro: class

Esta restricción especifica que el tipo parámetro debe ser un tipo por referencia: clase, array o delegado.

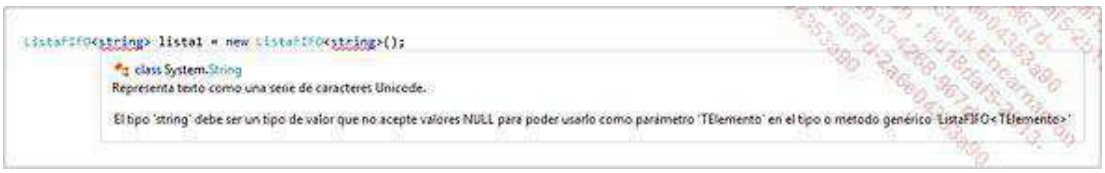
```
public class ListaFIFO<TElemento> where TElemento : class
{
}
```



where TParametro: struct

Esta restricción impone el uso de un tipo valor (y, por tanto, nullable) como tipo parámetro.

```
public class ListaFIFO<TElemento> where TElemento : struct
{
}
```



where TParametro: new()

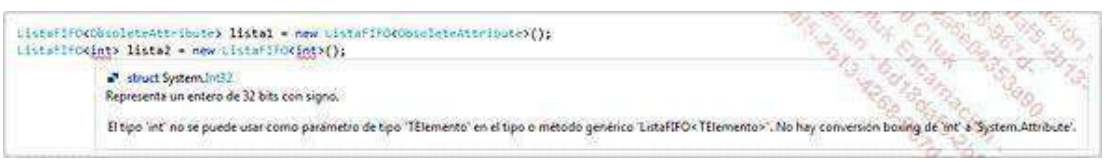
Cuando se aplica esta restricción, el tipo parámetro afectado debe tener un constructor sin parámetro público. Se utiliza cuando la clase genérica debe poder instanciar objetos del tipo parámetro.

```
public class ListaFIFO<TElemento> where TElemento : new()
{
}
```

where TParametro: <tipo>

Esta restricción indica que el tipo parámetro afectado debe ser del tipo especificado o derivar de él.

```
public class ListaFIFO<TElemento> where TElemento : Attribute
{
}
```



Cuando el tipo genérico posee varios parámetros es perfectamente viable definir una restricción sobre uno de ellos indicando que su tipo debe ser o debe derivar de otro tipo parámetro.

```
public class ParejaDeValores<TValor1, TValor2> where TValor2 :
```

```
TValor1
{
}
```

where TParametro: <interfaz>

Esta última restricción fuerza el uso de un tipo parámetro que implementa una interfaz particular.

```
public class ListaFIFO<TElemento> where TElemento : IComparable
{
}
```

Combinar restricciones

Es posible combinar las restricciones de manera que permitan obtener un control más fino sobre los tipos parámetros.

```
public class ListaFIFO<TElemento> : IListaFIFO<TElemento> where
TElemento : class, IComparable, new()
{
}
```

Algunas restricciones son compatibles entre sí o pueden utilizarse una única vez por tipo parámetro. Cuando se detecta un problema se produce un error de compilación.

4. Métodos

Los métodos genéricos son métodos que poseen uno o varios tipos parámetro. Es posible, también, realizar el mismo procesamiento sobre varios tipos de datos.

Estos métodos pueden definirse en tipos genéricos o no, lo cual puede resultar práctico para escribir métodos genéricos útiles ubicados en clases estáticas no genéricas.

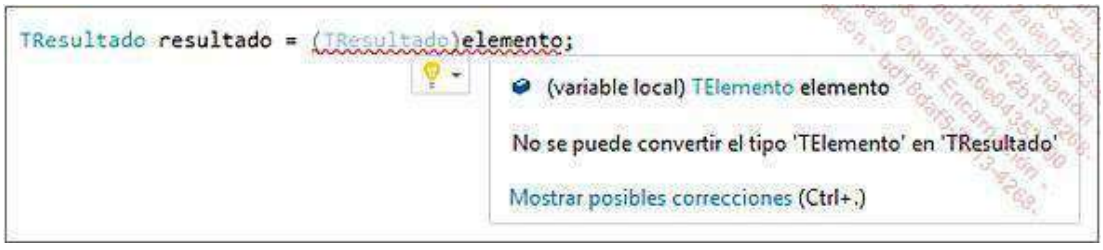
a. Definición de un método genérico

Como con la definición de una clase o de una interfaz, los métodos genéricos son muy similares a su contraparte no genérica. La diferencia reside en que agregan tipos parámetros a su firma.

La clase `ListaFIFO` puede, también, proveer un método genérico `Eliminar Elemento<TResultado>()` que devuelva un elemento con conversión del tipo parámetro de la clase hacia el tipo parámetro del método.

```
public TResultado EliminarElemento<TResultado>()
{
    //El alias de tipo TElemento se define a nivel de la clase
    //Se utiliza el método no genérico EliminarElemento para recuperar
    //un elemento de tipo TElemento
    TElemento elemento = EliminarElemento();

    //Se realiza una conversión de tipo utilizando el alias de tipo TResultado
    //como tipo "de destino"
    TResultado resultado = (TResultado)elemento;

    return resultado;
}
```

A pesar de su apariencia, sencilla, el código de este método genera un error de compilación a nivel de conversión de tipo:



Este error significa que el compilador no conoce ningún punto común entre los dos tipos que permita realizar una conversión. Para resolver este problema conviene agregar una restricción sobre el método genérico para especificar que `TResultado` es un tipo derivado de `TElemento`, lo que permite al compilador validar la precisión de la conversión de tipo.

```
public TResultado EliminarElemento<TResultado>() where TResultado :
TElemento
{
    TElemento elemento = EliminarElemento();
    TResultado resultado =  (TResultado)elemento;

    return resultado;
}
```

b. Uso de un método genérico

Los métodos genéricos se utilizan de la misma manera que los métodos normales. Como todos los elementos genéricos, basta con agregarles uno o varios tipos parámetros para poder utilizarlos.

```
//Instanciación de una lista de objetos.
var lista = new ListaFIFO<object>();
//Agregar un elemento de tipo string
lista.AgregarElemento("cadena 1");

//Recuperación del objeto bajo la forma de un string mediante
//el método genérico EliminarElemento<TResultado>
string cadena =  lista.EliminarElemento<string>();

Console.WriteLine("Mi objeto es la cadena {0}'", cadena);
```

5. Eventos y delegados

De la misma manera que es posible producir alertas cuando los hilos de espera se vuelven demasiado largos en las cajas de un supermercado, puede resultar interesante producir eventos cuando el número de elementos se hace demasiado grande en nuestra lista. Aquí, el evento se producirá cuando la lista contenga más de diez elementos.

En primer lugar, es preciso definir un delegado que recibirá como parámetro el elemento agregado que haya producido la alerta.

```
public delegate void ControladorAlertaNúmero<TElementoAgregado>(
TElementoAgregado últimoElementoAgregado)
```

El evento se declara de la siguiente forma:

```
public event ControladorAlertaNúmero<TElemento>
AlertaNúmeroElementos;
```

El tipo `TElemento` utilizado en la declaración de este evento es el que se utiliza como tipo parámetro para la clase. El tipo utilizado en el controlador de eventos estará, de este modo, adaptado al objeto `ListaFIFO` instanciado.

Una vez creado el evento es preciso desencadenarlo cuando el número de elementos se vuelva importante. Para ello, se define un método cuyo objeto es producir el evento si existe algún controlador y está asociado a él y se modifica el código del método `AgregarElemento` para invocarlo si es necesario.

```
protected virtual void ProducirAlertaNúmero(TElemento
últimoElementoAgregado)
{
    ControladorAlertaNúmero<TElemento> handler =
AlertaNúmeroElementos;

    //Si se ha definido uno o varios controladores,
    //handler no será null
    if (handler != null)
    {
           handler(últimoElementoAgregado);
    }
}
```

Por último, podemos asociar un controlador a una instancia de la lista y verificar que se produce el evento.

```
static void Main(string[] args)
{
    ListaFIFO<string> lista = new ListaFIFO<string>();
    //Agregar un controlador de eventos
    lista.AlertaNúmeroElementos += lista_AlertaNúmeroElementos;

    for (int i = 0; i < 10; i++)
    {
        lista.AgregarElementos("cadena " + 1);
    }
    Console.WriteLine("Existen 10 elementos en la lista.");
    Console.WriteLine("Agregar un elemento perturbador...");
    lista.AgregarElemento("elemento perturbador");

    Console.ReadLine();
}

static void lista_AlertaNúmeroElementos(string últimoElementoAgregado)
    => Console.WriteLine("¡Hay más de 10 elementos en la lista!
El último es '{0}'", últimoElementoAgregado);
```

Este código muestra el siguiente resultado:

```
Existen 10 elementos en la lista.
Agregar un elemento perturbador...
¡Hay más de 10 elementos en la lista! El último es
'elemento perturbador'
```

Las colecciones

Es habitual que una aplicación tenga que manipular grandes cantidades de datos. Para ello, el framework .NET provee varias estructuras de datos, agrupados bajo el nombre de colecciones. Estas están adaptadas a distintos tipos de situación: un almacenamiento desordenado de datos dispares, el almacenamiento de datos en base a su tipo, el almacenamiento de datos por nombre, etc.

1. Tipos existentes

Las distintas clases que permiten gestionar colecciones se agrupan en dos espacios de nombres:

- `System.Collections`
- `System.Collections.Generic`

El primero representa a los tipos "clásicos", mientras que el segundo incluye las clases genéricas equivalentes que permiten trabajar con objetos fuertemente tipados.

a. Array

La clase `Array` no se encuentra en el espacio de nombres `System.Collections`, pero puede considerarse como una colección. En efecto, implementa varias interfaces propias de las colecciones: `IList`, `ICollection` e `IEnumerable`. Esta clase es la clase de base para todos los arrays utilizados en C#.

No obstante, a menudo se utiliza esta clase directamente: se la prefiere en la mayoría de casos en la sintaxis C#.

La clase `Array`, abstracta, no permite instanciarla mediante el operador `new`. Por este motivo, se utilizará alguna de las sobrecargas del método estático `Array.CreateInstance`.

```
Array array = Array.CreateInstance(typeof(int), 5);
```

Esta declaración de variable es equivalente a la siguiente:

```
int[] array = new int[5];
```

b. ArrayList y List<T>

Las clases `ArrayList` y su contraparte genérica `List<T>` son evoluciones de la clase `Array`. Aportan ciertas mejoras respecto a los arrays:

- El tamaño de un objeto `ArrayList` o `List<T>` es dinámico y se ajusta en función de las necesidades.
- Estas clases aportan métodos para agregar, incluir o eliminar varios elementos.

En cambio, las listas tienen una única dimensión, lo que puede complicar ciertos procesamientos.

Las colecciones de tipo `ArrayList` también pueden presentar problemas de rendimiento, pues manipulan elementos de tipo `Object`, lo que implica muchas conversiones de tipo. Es preferible utilizar arrays fuertemente tipados o listas genéricas (`List<T>`) que aseguran, también, un tipado fuerte.

La clase `ArrayList` puede instanciarse mediante uno de sus tres constructores. El primero no recibe ningún parámetro: el objeto creado tendrá, también, una capacidad inicial igual a cero y se dimensionará automáticamente cada vez que se agregue un elemento. Este dimensionamiento automático es costoso en términos de recursos y es preferible, siempre que se conozca el tamaño final de la lista, utilizar el segundo constructor que recibe como parámetro un valor numérico que indique la capacidad inicial del objeto. El tercer y último constructor permite instanciar un objeto `ArrayList` proporcionando una colección origen, a partir de la cual debe completarse.

Durante su uso es posible obtener el número de elementos en un `ArrayList` utilizando la propiedad `Count`. La propiedad `Capacity` devuelve el número de elementos que puede contener la colección en su estado actual. Estos dos valores varían en función de los elementos que se agregan y eliminan.

El siguiente código muestra el funcionamiento de la clase `ArrayList`:

```
static void Main(string[] args)
{
    ArrayList lista = new ArrayList();
    Console.WriteLine("Número de elementos en la lista: {0}",
lista.Count);
    Console.WriteLine("Capacidad inicial de la lista: {0}",
lista.Capacity);
    Console.WriteLine();

    Console.WriteLine("Agregar 10 elementos en la lista");
    for (int i = 0; i < 10; i++)
    {
        string valor = "cadena " + i;
        lista.Add(valor);
    }

    Console.WriteLine("El número de elementos en la lista es
ahora {0}", lista.Count);
    Console.WriteLine("Su capacidad es ahora {0}",
lista.Capacity);

    Console.WriteLine("Los elementos presentes en la lista son:");
    foreach (string cadena in lista)
    {
        //El carácter \t es una tabulación que permite
        //indexar nuestra lista de elementos
        Console.WriteLine("\t{0}", cadena);
    }
    Console.WriteLine();

    Console.WriteLine("Agregar 10 nuevos elementos en la lista");
    for (int i = 10; i < 20; i++)
    {
        string valor = "cadena " + i;
        lista.Add(valor);
    }

    Console.WriteLine("El número de elementos en la lista es
ahora {0}", lista.Count);
```

```
    Console.WriteLine("Su capacidad es ahora {0}",
lista.Capacity);
    Console.WriteLine();

    string cadenaIndice3 = (string)lista[3];
    string cadenaIndice4 = (string)lista[4];
    string cadenaIndice5 = (string)lista[5];
    string cadenaIndice6 =  (string)lista[6];
    Console.WriteLine("La cadena del índice 3 de la lista es
'{0}'", cadenaIndice3);
    Console.WriteLine("La cadena del índice 4 de la lista es
'{0}'", cadenaIndice4);
    Console.WriteLine("La cadena del índice 5 de la lista es
'{0}'", cadenaIndice5);
    Console.WriteLine("La cadena del índice 6 de la lista es
'{0}'", cadenaIndice6);
    Console.WriteLine();

    Console.WriteLine("Eliminar los elementos en los índices 3 y 4
    lista.RemoveRange(3, 4);

    cadenaIndice3 = (string)lista[3];
    cadenaIndice4 = (string)lista[4];
    cadenaIndice5 = (string)lista[5];
    cadenaIndice6 = (string)lista[6];
    Console.WriteLine("Las cadenas en los índices 3, 4, 5 y 6 son
ahora '{0}', '{1}', '{2}' y '{3}'", cadenaIndice3,
cadenaIndice4, cadenaIndice5, cadenaIndice6);
    Console.WriteLine();

    Console.WriteLine("Insertar la cadena 'elemento insertado' en
el índice 3");
    lista.Insert(3, "elemento insertado");

    cadenaIndice3 = (string)lista[3];
    cadenaIndice4 = (string)lista[4];
    cadenaIndice5 = (string)lista[5];
    cadenaIndice6 = (string)lista[6];
    Console.WriteLine("Las cadenas en los índices 3, 4, 5 y 6 son
ahora '{0}', '{1}', '{2}' y '{3}'", cadenaIndice3,
cadenaIndice4, cadenaIndice5, cadenaIndice6);
    Console.WriteLine();

    Console.WriteLine("Eliminar todos los elementos de la lista");
    lista.Clear();

    Console.WriteLine("El número de elementos en la lista es
ahora {0}", lista.Count);
    Console.WriteLine("Su capacidad es ahora {0}",
lista.Capacity);

    Console.ReadLine();
}
```

Muestra el siguiente resultado:

```
Agregar 10 elementos en la lista
El número de elementos en la lista es ahora 10
Su capacidad es ahora 16
Los elementos presentes en la lista son:
        cadena 0
        cadena 1
        cadena 2
        cadena 3
        cadena 4
        cadena 5
        cadena 6
        cadena 7
        cadena 8
        cadena 9

Agregar 10 nuevos elementos en la lista
El número de elementos en la lista es ahora 20
Su capacidad es ahora 32

La cadena del índice 3 de la lista es 'cadena 3'
La cadena del índice 4 de la lista es 'cadena 4'
La cadena del índice 5 de la lista es 'cadena 5'
La cadena del índice 6 de la lista es 'cadena  6'

Eliminar los elementos en los índices 3, 4, 5 y 6
Las cadenas en los índices 3, 4, 5 y 6 son ahora 'cadena 7',
'cadena 8', 'cadena 9' y 'cadena 10'

Insertar la cadena 'elemento insertado' en el índice 3
Las cadenas en los índices 3, 4, 5 y 6 son ahora 'elemento
insertado', 'cadena 7', 'cadena 8' y 'cadena 9'

Eliminar todos los elementos de la lista
El número de elementos en la lista es ahora 0
Su capacidad es ahora 32
```

El código anterior funciona exactamente de la misma manera remplazando la línea:

```
ArrayList lista = new ArrayList();
```

por la siguiente línea que utiliza la colección genérica `List<T>`:

```
List<string> lista = new List<string>();
```

Para aprovechar al máximo esta lista genérica, se recomienda eliminar todas las conversiones de tipo que permiten leer los valores en la lista. No son necesarios, pues el objeto `List<string>` devuelve objetos de tipo `string`.

```
string cadenaIndice3 = (string)lista[3];
```

se convierte en:

```
string cadenaIndice3 = lista[3];
```

c. Hashtable y Dictionary<TKey, TValue>

Las clases `Hashtable` y su equivalente genérico `Dictionary<TKey, TValue>` permiten almacenar objetos proporcionando, para cada uno, un identificador único. Puede ser una cadena de caracteres, un valor numérico, un objeto `Type` o cualquier otro objeto. Los datos pueden, a continuación, recuperarse mediante su identificador gracias a los índices implementados en ambas colecciones.

Estos tipos también implementan el método `ContainsKey` que permite saber si un identificador existe en la colección. Este método puede resultar muy práctico, pues un acceso sobre una clave que no exista producirá una excepción.

El siguiente código muestra el funcionamiento de la clase `Hashtable`:

```
static void Main(string[] args)
{
    Hashtable colección = new Hashtable();
    Console.WriteLine("El número de elementos inicial es {0}",
colección.Count);
    Console.WriteLine();

    Console.WriteLine("Agregar 10 elementos");
    for (int i = 0; i < 10; i++)
    {
        colección.Add(i, "cadena " + i);
    }

    Console.WriteLine("El número de elementos es {0}",
colección.Count);
    Console.WriteLine();

    Console.WriteLine("Los elementos en la colección son:");
    foreach (int identificador in colección.Keys)
    {
        string valor = (string) colección[identificador];
        Console.WriteLine("\tClave: {0} - Valor: {1}",
identificador, valor);
    }
    Console.WriteLine();

    Console.WriteLine("Eliminar el elemento cuya clave es 3");
    colección.Remove(3);

    string claveExiste;
    if (colección.ContainsKey(3))
```

```
    {
        claveExiste = "SI";
    }
    else
    {
        claveExiste = "NO";
    }
    Console.WriteLine("¿Existe la clave 3? {0}", claveExiste);
    Console.WriteLine("El número de elementos es {0}", colección.Count);
    Console.WriteLine();

    Console.WriteLine("Eliminar todos los elementos");
    collection.Clear();
    Console.WriteLine("El número de elementos es {0}", colección.Count);

    Console.ReadLine();
}
```

Este código produce el siguiente resultado:

```
El número de elementos inicial es 0

Agregar 10 elementos
El número de elementos es 10

Los elementos en la colección son:
        Clave: 9 - Valor: cadena  9
        Clave: 8 - Valor: cadena  8
        Clave: 7 - Valor: cadena  7
        Clave: 6 - Valor: cadena  6
        Clave: 5 - Valor: cadena  5
        Clave: 4 - Valor: cadena  4
        Clave: 3 - Valor: cadena  3
        Clave: 2 - Valor: cadena  2
        Clave: 1 - Valor: cadena  1
        Clave: 0 - Valor: cadena  0

Eliminar el elemento cuya clave es 3
¿Existe la clave 3? NO
El número de elementos es  9

Eliminar todos los elementos
El número de elementos es  0
```

Este código puede adaptarse fácilmente para utilizar un objeto de tipo `Dictionary` en lugar del objeto `HashTable`.

Basta con remplazar la línea:

```
Hashtable colección = new Hashtable();
```

por la línea siguiente:

```
Dictionary<int, string> colección = new Dictionary<int, string>();
```

El tipo `Dictionary`, al ser fuertemente tipado, hace que ya no sea necesario realizar una conversión de tipo para recuperar el valor de tipo `string`. La línea:

```
string valor = (string)colección[identificador];
```

se convierte en:

```
string valor = colección[identificador];
```

d. Stack y Stack<T>

Las pilas (*stack*, en inglés) son colecciones que siguen el principio LIFO (*Last In, First Out*): solamente es posible acceder al último elemento de la colección, y cuando se agrega un elemento, este se sitúa automáticamente en la última posición de la colección.

Existe una analogía común para representar estos objetos: una pila de platos. Cada plato que se agrega se sitúa en lo alto de la pila y cuando se desea tomar un plato de la pila, se toma el que se encuentra por encima de todos los demás. El último plato insertado es el primero en salir de la pila.

Las tres operaciones principales disponibles para estos dos tipos son:

- `Push`, para agregar un elemento en lo alto de la pila.
- `Peek`, para tomar el elemento de lo alto de la pila sin eliminarlo de la colección.
- `Pop`, para recuperar el último elemento de la pila eliminándolo de la colección.

e. Queue y Queue<T>

Las colas (*queue*, en inglés) siguen el mismo principio que las pilas, pero con una diferencia importante: las colas siguen el principio FIFO (*First In, First Out*), lo que quiere decir que el primer elemento agregado será el primero en salir de la cola.

La analogía evidente que podemos utilizar para ilustrar este principio es una cola de espera: la primera persona que llega es la primera en salir de la cola de espera para ser atendida en una caja, por ejemplo.

Como las clases `Stack` y `Stack<T>`, los tipos `Queue` y `Queue<T>` implementan tres métodos principales:

- `Enqueue`, para agregar un elemento al final de la cola.
- `Peek`, para recuperar el primer elemento de la cola sin eliminarlo de la colección.
- `Dequeue`, para recuperar el primer elemento de la cola eliminándolo de la colección.

2. Seleccionar un tipo de colección

Es muy importante identificar las necesidades relativas a una colección de objetos antes de escoger un tipo u otro. En efecto, la estructura de una clase entera, o incluso más, puede depender de un tipo de colección.

He aquí una lista de aspectos a considerar de cara a seleccionar el tipo que mejor se adapte a nuestras necesidades:

- Si es necesario acceder mediante el índice a un elemento de la colección, opte por un array, un `ArrayList` o una `List<T>`.
- Si desea almacenar parejas del tipo "clave única & elemento", utilice una `Hashtable` o un `Dictionary<TKey,TValue>`.
- Si debe acceder a los elementos en el mismo orden en el que se han insertado en la colección, utilice una `Queue` o su equivalente genérico. Si desea acceder a los elementos en el orden inverso, utilice una `Stack` o una `Stack<T>`.
- Si debe almacenar elementos ordenados, utilice las clases `ArrayList` o `Hashtable`, o eventualmente las colecciones más avanzadas `SortedSet`, `SortedList` o `SortedList<Tkey, TValue>`.

Programación dinámica

Desde sus inicios, C# es un lenguaje con un tipado fuerte y estático, lo que significa que el compilador es capaz de conocer el tipo de cada variable que se utiliza en una aplicación. Esto le permite saber cómo tratar las llamadas a los métodos y las propiedades de cada objeto y permite generar errores de compilación cuando se detecta que alguna operación es inválida.

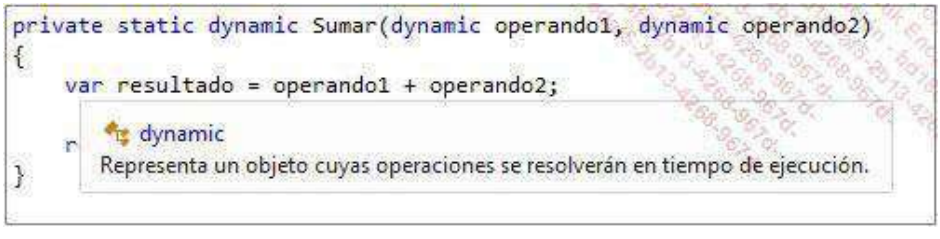
Con la aparición de C# 4 y del entorno .NET en versión 4.0 se introduce un nuevo tipo de dato singular: `dynamic`. Este tipo de dato está destinado a resolver problemáticas muy particulares, pues permite utilizar variables cuyo tipo no se conoce en tiempo de ejecución.

El compilador no realiza ninguna verificación para aquellas operaciones realizadas sobre variables `dynamic`. Se realizan únicamente en tiempo de ejecución de la aplicación: cualquier operación que se detecte como inválida en este punto genera un error de ejecución.

La siguiente función recibe dos parámetros de tipo `dynamic` y les aplica el operador +.

```
private static dynamic Sumar(dynamic operando1, dynamic
operando2)
{
    var resultado = operando1 + operando2;

    return resultado;
}
```

El tipo de retorno de la función es `dynamic`, pues no es posible determinarlo en tiempo de compilación. El uso de la inferencia de tipo en Visual Studio lo muestra perfectamente si situamos el cursor del ratón sobre la palabra clave `var`:



Los operandos, al ser de tipo `dynamic`, hacen que IntelliSense sea incapaz de proponer métodos o propiedades para su uso.

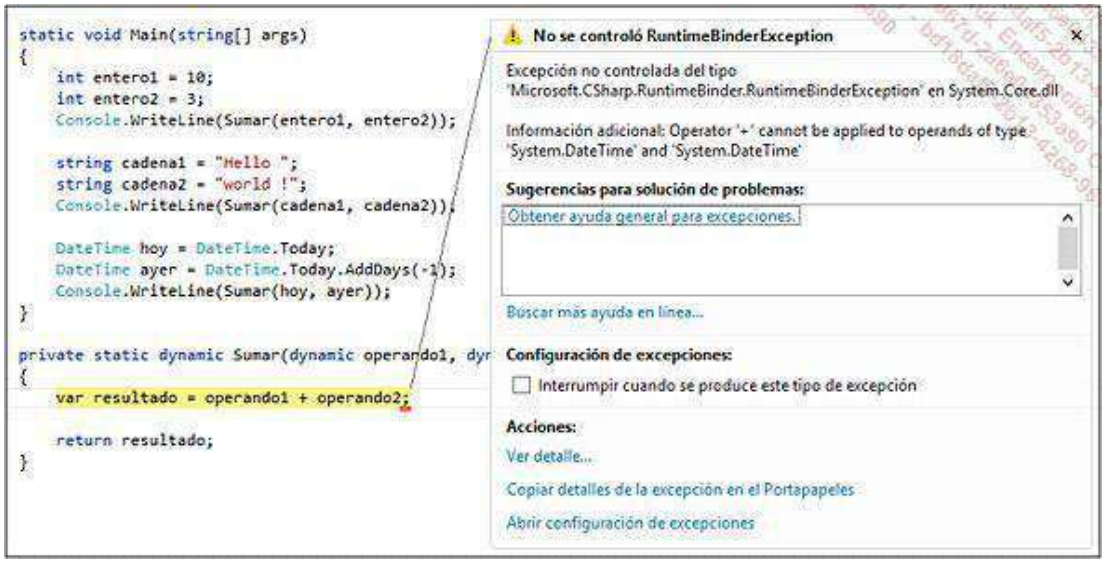
El siguiente código permite verificar el funcionamiento del método `Sumar`:

```
int entero1 = 10;
int entero2 = 3;
Console.WriteLine(Sumar(entero1, entero2));

string cadena1 = "Hello ";
string cadena2 = "world !";
Console.WriteLine(Sumar(cadena1, cadena2));

DateTime hoy = DateTime.Today;
DateTime ayer = DateTime.Today.AddDays(-1);
```

```
Console.WriteLine(Sumar(hoy, ayer));
```

Este código compila sin problema, pero podemos avanzar un inconveniente: es imposible sumar dos objetos de tipo `DateTime`.

La ejecución de este código no produce ningún error en las dos primeras sumas. En cambio, como era de esperar, la operación que utiliza dos objetos de tipo `DateTime` genera un error en tiempo de ejecución:



Conviene, por tanto, ser especialmente prudente con el uso del tipo `dynamic`, podría generar errores imposibles de detectar en tiempo de compilación.

En la práctica, este tipo se utiliza a menudo en contextos de interoperabilidad con objetos COM o con lenguajes dinámicos (IronPython o IronRuby, por ejemplo).

El uso de objetos dinámicos implica una pérdida de rendimiento durante la ejecución. Conviene, por tanto, utilizar el tipo `dynamic` únicamente cuando sea realmente indispensable para realizar un procesamiento.

Programación asíncrona

Cada vez es más frecuente tener que desarrollar aplicaciones reactivas y capaces de realizar varias tareas simultáneamente. El framework .NET aporta soluciones a este problema desde sus inicios mediante las clases `Thread` o `BackgroundWorker`, entre otras. Con el framework .NET 4.0 aparecen la clase `Task` y su equivalente genérico `Task<TResult>`. Estos tipos simplifican el trabajo del desarrollador permitiéndole gestionar de manera sencilla el procesamiento y la espera de la ejecución de bloques de código, proporcionando también un medio para ejecutar varios procesamientos asíncronos a la vez.

Con la llegada de C# 5 la tarea del desarrollador se simplifica todavía más gracias a la integración del asincronismo directamente en el lenguaje. En efecto, las palabras clave `async` y `await` permiten escribir código asíncrono de manera secuencial, como código... ¡síncrono!

1. Los objetos Task

Las clases `Task` y `Task<TResult>` permiten ejecutar código asíncrono encapsulando el uso de threads.

Funcionamiento de los threads

Los threads son unidades de ejecución que pueden trabajar, en función de la arquitectura de la máquina, en paralelo (cada thread se ejecuta durante un pequeño espacio de tiempo y a continuación cede su lugar a otro thread, que cederá de nuevo su espacio a otro thread, etc.). Será el sistema operativo el que decidirá el tiempo de procesador dedicado a cada thread. Por este motivo, es imposible prever exactamente en qué momento se ejecutará una instrucción determinada situada dentro de un thread.

Creación de un objeto Task

Antes de cualquier uso, es necesario instanciar los objetos `Task`. Para ello, tenemos ocho constructores a nuestra disposición, y todos ellos reciben como primer parámetro un delegado (de tipo `Action` o `Func`). Este delegado se corresponde con la sección de código que debe ejecutarse en un thread separado. El procesamiento se lanza, a continuación, mediante el método `Start` del objeto `Task`.

```
static void Main(string[] args)
{
    Console.WriteLine("Este código se ejecuta en el thread
principal cuyo identificador es {0}",
Thread.CurrentThread.ManagedThreadId);

    Action actionAEjecutar = () => { Console.WriteLine("Este código
se ejecuta en un thread separado cuyo identificador es {0}",
Thread.CurrentThread.ManagedThreadId); };

    Task task = new Task(actionAEjecutar);
    Task task2 = new  Task(actionAEjecutar);

    task.Start();
    task2.Start();

    Console.ReadLine();
}
```

El mismo código se ejecuta, aquí, dos veces, en dos threads distintos. El resultado de esta sección de código es el siguiente:

```
Este código se ejecuta en el thread principal cuyo identificador
es 10
Este código se ejecuta en un thread separado cuyo identificador
es 11
Este código se ejecuta en un thread separado cuyo identificador
es 12
```

Tras una nueva ejecución, el resultado no será, necesariamente, el mismo. En efecto, se crean dos nuevos threads y el valor de la propiedad `Thread.CurrentThread.ManagedThreadId` relativo a cada uno de los threads será, probablemente, distinto.

El uso de la clase `Task<TResult>` permite, por su parte, recuperar un resultado del procesamiento fuertemente tipado. El siguiente código suma los números enteros de 1 a 10.000 en un thread separado y devuelve el resultado. Este resultado se muestra mediante la propiedad `Result`de la clase `Task <Tresult>`.

```
static void Main(string[] args)
{
    Procesamiento();
    Console.ReadLine();
}

private static void Procesamiento()
{
    Task<long> tareaCálculo = new Task<long>(() =>
        {
            long result = 0;
            for (long i = 1; i <= 10000; i++)
                result += i;

            return result;
        });

    tareaCálculo.Start();
    Console.WriteLine(tareaCálculo.Result);
}
```

Esta propiedad espera, de manera síncrona, a que finalice la ejecución del procesamiento para mostrar su resultado. Para mantener la noción de asincronismo, es posible utilizar el método `ContinueWith`, que crea una nueva tarea pasándole como parámetro de entrada la tarea anterior.

```
private static void Procesamiento()
{
    Task<long> tareaCálculo = new Task<long>(() =>
        {
            long result = 0;
            for (long i = 1; i <= 10000; i++)
                result += i;
```

```
            return result;
        });

    tareaCálculo.ContinueWith(tareaAnterior =>
Console.WriteLine(tareaAnterior.Result));

    tareaCálculo.Start();
}
```

2. Escribir código asíncrono con async y await

La palabra clave `await` permite esperar a que finalice la ejecución de una `Task` iniciada. Durante esta espera, la aplicación cede el control a los threads que necesitan tiempo de procesador, en particular para mantener la interfaz gráfica reactiva. El código situado a continuación de una instrucción que utilice esta palabra clave se ejecutará una vez finalice el procesamiento asíncrono, como si estuviera situado en un método `ContinueWith`.

La tarea que deba esperarse puede ser de tipo `Task` o `Task<T>`. Este último tipo se utiliza cuando la tarea devuelve un valor. Si el valor es de tipo `int`, entonces la tarea debe ser de tipo `Task<int>`.

```
//Al final de la ejecución, la variable resultado contiene 1
int resultado = await Task<int>.Run(() => return  1);
```

Cualquier método que contenga una instrucción que utilice `await` debe utilizar la palabra clave `async` en su declaración. Es posible, potencialmente, invocar a un método marcado con esta palabra clave mediante la palabra clave `await`, aunque para ello es preciso modificar la firma de este método asíncrono de manera que devuelva un objeto de tipo `Task` o `Task<T>`.

El siguiente código es una modificación del código que hemos visto antes. El método `Procesamiento` no devuelve aquí ningún objeto `Task` puesto que se invoca desde el método `Main` de la aplicación. De modo que está estrictamente prohibido utilizar las palabras clave `async` y `await` en este método.

```
private static async void Procesamiento()
{
    long resultado = await Task<long>.Run(() =>
        {
            long result = 0;
            for (long i = 1; i <= 10000; i++)
                result += i;

            return result;
        });

    Console.WriteLine(resultado);
}
```

Los distintos tipos de errores

Ningún software está exento de errores y ningún desarrollador escribe código sin cometer la más mínima falta de sintaxis u olvidarse de un punto y coma. A pesar de toda la atención que pueda prestar un desarrollador en su código, estos errores son inevitables. Es necesario, por tanto, poder identificarlos y corregirlos.

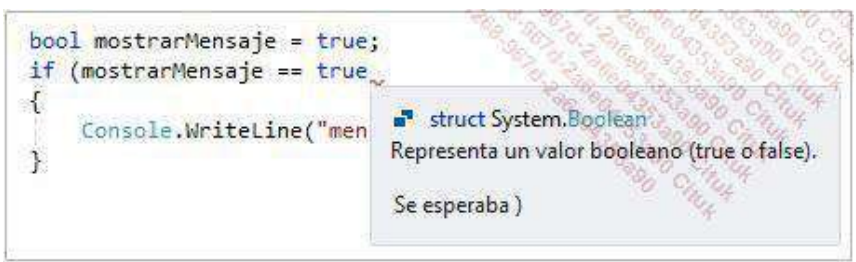
1. Errores de compilación

Los errores de compilación son errores que impiden la generación de un ensamblado a partir del código. Estos errores son, a menudo, errores de sintaxis, pero pueden también estar vinculados a un mal uso de un objeto, de un método o de cualquier otro elemento del código.

Estos errores se detectan en tiempo de compilación, evidentemente, pero también durante la escritura de un programa. En efecto, Visual Studio analiza el código de manera permanente para proveer la máxima cantidad de información lo antes posible, lo que disminuye la dificultad y el coste de corregir cada uno de los errores.

Visualización en la ventana de edición

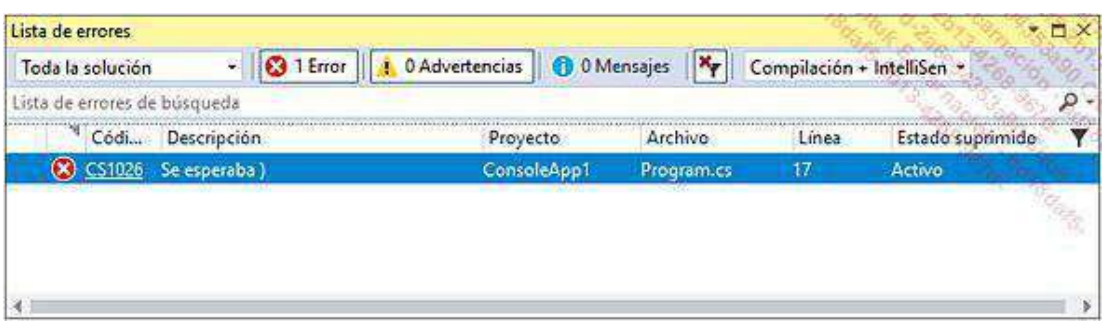
Los errores detectados por el análisis permanente se muestran directamente en la ventana de edición. Las secciones de código que generan errores están marcadas con un subrayado de color rojo. Es posible ver, también, un cuadro con información detallada acerca del error sobre cada una de estas secciones.



Ventana Lista de errores

Los errores detectados en tiempo de compilación se muestran en la ventana Lista de errores. Esta ventana puede abrirse mediante el menú **Ver** y, a continuación, **Lista de errores**.

Por cada uno de los errores encontrados, se muestra información que permite su localización, con el objetivo de permitir una corrección más rápida.



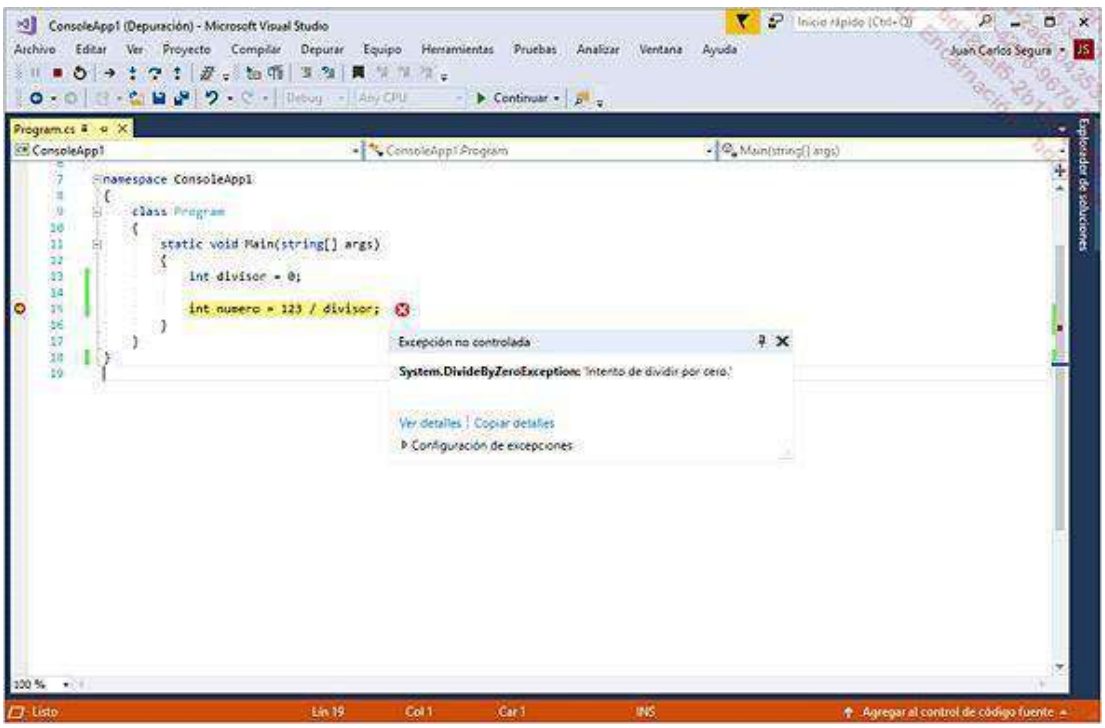
Al hacer doble clic sobre cada uno de los errores reportados en esta ventana se abre el editor de código en el lugar correspondiente.

2. Errores de ejecución

Estos errores son aquellos que pueden producirse una vez generado el ensamblado y ejecutado el código. A estos errores se les denomina **excepciones**.

Este tipo de error puede producirse debido a un entorno de ejecución que no se corresponda con las condiciones esperadas o por un error lógico. La causa puede, por tanto, ser un archivo que no existe, la saturación de la memoria de la máquina o la búsqueda del duodécimo elemento de un array que contiene solamente once.

En fase de depuración de la aplicación, Visual Studio indica de manera particularmente explícita estos errores. En efecto, cuando se produce una excepción, el depurador pone en pausa la ejecución de la aplicación y muestra una ventana que incluye la descripción del error.



Uso de excepciones

La generación de una excepción permite interrumpir el flujo normal de ejecución de la aplicación proporcionando información acerca del error que representa. Sin intervención alguna por parte del desarrollador para gestionarla, se propaga automáticamente a través de la pila de llamadas hasta el punto de entrada de la aplicación e incluso puede ocasionar su cierre.

1. Creación y generación de excepciones

Las excepciones se representan mediante objetos que heredan de la clase `System.Exception`. Pueden generarse (producirse) en cualquier lugar de la aplicación.

a. La clase Exception

La clase `System.Exception` es la clase de base de la que heredan todas las excepciones. Sus miembros le permiten contener mucha información relativa al error en curso de propagación. La mayoría de estos miembros obtienen valor automáticamente cuando se produce una excepción. Entre ellos encontramos, en particular, la propiedad `StackTrace` que contiene información particularmente interesante para localizar el origen de la excepción.

La instanciación de un objeto de tipo `Exception` puede llevarse a cabo mediante tres constructores:

```
public Exception();

public Exception(string message);

public Exception(string message, Exception innerException);
```

El parámetro `message` se corresponde con una etiqueta personalizada que permite comprender el error cuando se está tratando lejos de su origen. Es posible recuperar este mensaje más adelante consultando la propiedad `Message` de la excepción.

El parámetro `innerException` permite encapsular una excepción que se está tratando en una nueva excepción generada.

b. La palabra clave throw

Instanciar un objeto de tipo `Exception` no basta para producir una excepción. Para ello, es necesario ejecutar una instrucción `throw` proporcionando el objeto `Exception` que se desea elevar.

```
public void ContarHasta(int número)
{
    if (número < 0)
    {
        string mensaje = string.Format("Es imposible contar
de 0 a {0}", número);
        Exception error = new Exception(mensaje);
        throw error;
    }
```

```
    //Este código solo se ejecutará si no se produce ninguna excepción
    for (int i = 0; i <número; i++)
    {
       Console.WriteLine(i);
    }
}
```

Si el valor del parámetro `número` es inferior a 0 se produce una excepción que interrumpe el flujo de ejecución normal del programa. Podrá gestionarse en el método que invoca al procedimiento `ContarHasta` o más adelante en la pila de llamadas.

c. Excepciones especializadas

En ciertos casos, puede resultar interesante producir excepciones específicas a un tipo de comportamiento. Esto puede permitirnos identificar muy rápidamente errores sencillos de corregir, pero poco evidentes de identificar o localizar la zona de la aplicación en la que se produce la excepción, o incluso proveer información particular para tratar el error.

El framework .NET provee un cierto número de tipos de excepciones adaptados a circunstancias particulares. A continuación se muestran algunos de estos tipos:

`NotImplementedException`

Se produce, por defecto, en todos los esqueletos de código generado automáticamente por Visual Studio. Significa que una sección de código no se ha implementado todavía.

`StackOverflowException`

La produce el CLR cuando el número de elementos en la pila de llamadas se hace demasiado importante. Se produce típicamente cuando se realiza una llamada recursiva.

`SqlException`

La producen las clases de acceso a bases de datos SQL Server, en particular cuando una consulta SQL es incorrecta.

`DivideByZeroException`

Se produce cuando se realiza un cálculo que intenta realizar una división entre cero.

`OutOfMemoryException`

Se produce cuando la aplicación necesita un espacio de memoria que el sistema no está en posición de proveer.

2. Gestionar las excepciones

Las excepciones pueden causar que finalice la ejecución de una aplicación y es conveniente que el desarrollador sepa tratarlas convenientemente.

En efecto, a menudo ocurre que la ejecución de una aplicación puede continuar aun en caso de producirse algún

error, degradando eventualmente alguna funcionalidad si las circunstancias así lo exigen.

a. La estructura try ... catch

La única manera de interceptar una excepción en C# consiste en utilizar la estructura `try ... catch`. Esta estructura está formada por dos partes:

- Un bloque `try` que permite definir un juego de instrucciones "de riesgo".
- Uno o varios bloques `catch` que permiten especificar el código que se debe ejecutar en caso de producirse una excepción.

La sintaxis general de uso de esta estructura es la siguiente:

```
try
{
    //Juego de instrucciones de riesgo
}
catch (<tipo de excepción> <nombre de variable>)
{
    //Instrucciones que permiten tratar las excepciones
}
[ otro(s) bloque(s) catch ... ]
```

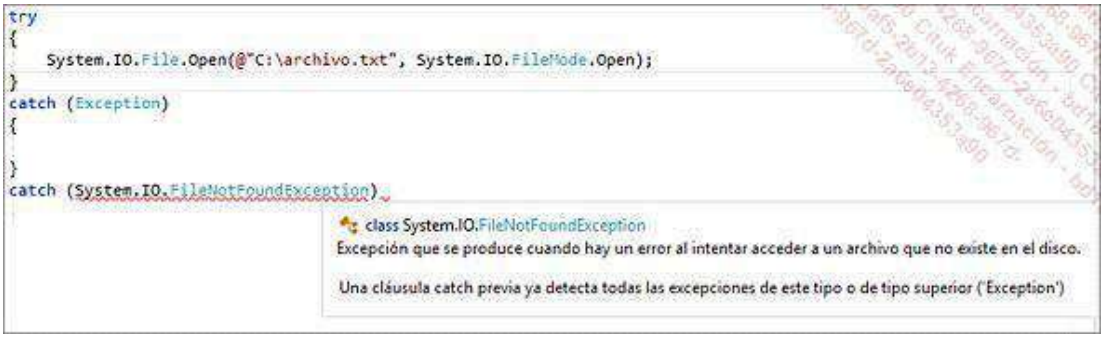
Cuando alguna de las instrucciones contenidas en el bloque `try {}` produce una excepción, el sistema examina cada uno de los bloques `{}` y ejecuta el código contenido en el primer bloque `catch` cuyo tipo de excepción se corresponda con el error generado.

En ese caso, una vez que el error se gestiona dentro del bloque `catch {}`, el método continúa con su ejecución a partir de la línea siguiente al final de la instrucción `try ... catch`.

La llamada a métodos asíncronos (mediante la palabra clave `async`) no es posible en los bloques `catch` salvo a partir de la versión 6 de C#.

Cuando es preciso gestionar distintos tipos de excepciones es importante reflexionar acerca del orden en el que declarar los distintos bloques. En efecto, si se implementan dos bloques que gestionan los tipos `Exception` y `FileNotFoundException`, declarados en este orden, el bloque que se ejecuta en el caso de una `FileNotFoundException` será el bloque... ¡que gestiona el tipo `Exception`!

Visual Studio es capaz, no obstante, de prevenirnos acerca de este problema:



Esto se explica fácilmente debido al hecho de que `FileNotFoundException` hereda del tipo `Exception`. La

excepción producida es, efectivamente, del tipo `Exception` y se gestiona, por tanto, dentro del primer bloque.

Para evitar este problema es preciso declarar los bloques correspondientes a los tipos más especializados en primer lugar y terminar por aquellos tipos menos especializados.

```
try
{
    System.IO.File.Open(@"C:\archivo.txt", System.IO.FileMode.Open);
}
catch (System.IO.FileNotFoundException)
{

}
catch (Exception)
{

}
```

b. Los filtros de excepción

En ciertos casos, la ejecución de un proceso relativo a una excepción solo debe producirse en determinadas condiciones. Para ello, la solución mejor adaptada es la inserción de comprobaciones en un bloque `catch`.

```
try
{
    //Generación de un número aleatorio
    var random = new Random();
    int numero = random.Next();

    Console.WriteLine("Al azar: {0}", numero);

    //Si el número es par
    if (numero % 2 == 0)
    {
        throw new Exception("El número es par");
    }
    //Si no
    else
    {
        throw new Exception("El número es impar");
    }
}
catch (Exception ex)
{
    if (ex.Message.Contains(" par"))
        Console.WriteLine("Tratamiento de la excepción: el número es par");
    else
        Console.WriteLine("Tratamiento de la excepción: el número es impar");
}
```

El ejemplo que se presenta aquí no incluye más que dos casos muy simples de procesamientos, aunque pueden ser numerosos. Los errores COM se presentan, en particular, bajo un único tipo de excepción: `COMException`. Es la propiedad `HResult` del objeto `COMException` la que contiene la información relativa al tipo de error producido. En este caso resulta muy sencillo encontrarse con un bloque `catch` muy largo y, por consiguiente,

difícil de mantener.

Los filtros de excepción ofrecen la posibilidad de distribuir el procesamiento de cada una de estas excepciones en un bloque `catch` mediante la palabra clave `when` seguida de un predicado que devuelva un valor booleano.

El ejemplo anterior podría reescribirse del siguiente modo para aprovechar esta funcionalidad introducida en C# 6:

```
try
{
    //Generación de un número aleatorio
    var random = new Random();
    int numero = random.Next();

    Console.WriteLine("Al azar: {0}", numero);

    //Si el número es par
    if (numero % 2 == 0)
    {
        throw new Exception("El número es par");
    }
    //Si no
    else
    {
        throw new Exception("El número es impar");
    }
}
catch (Exception ex) when (ex.Message.Contains(" par"))
{
    Console.WriteLine("Tratamiento de la excepción: el número es par");
}
catch (Exception ex) when (ex.Message.Contains("impar"))
{
    Console.WriteLine("Tratamiento de la excepción: el número es impar");
}
```

También es posible, evidentemente, combinar el uso de bloques `catch` "simples" con bloques que utilicen filtros de excepción para cubrir la máxima cantidad de errores de ejecución. Conviene tener especialmente en cuenta que solo puede ejecutarse un único bloque `catch`, tenga asociado un filtro o no.

c. El bloque finally

Existen algunos procesos que deben realizarse tras la ejecución correcta de una sección de código, pero también cuando se produce alguna excepción. Estos procesos se pueden corresponder, por lo general, con la liberación de recursos, una notificación al usuario o un registro en una base de datos.

Para ello, el uso de una estructura `try ... catch` puede resultar inconveniente dado que es necesario duplicar cierta parte del código: como mínimo una instrucción idéntica al final de los bloques `try` y `catch`. Esta situación, que no es demasiado óptima, puede resolverse utilizando un bloque `finally` que permite simplificar la escritura, la lectura y el mantenimiento del código.

Este tipo de bloque se agrega a continuación de una estructura `try ... catch`de la siguiente manera:

```
try
{
    //...
```

```
}
catch (...)
{
    //...
}
finally
{
    //Procesamientos a realizar sistemáticamente
}
```

También es posible agregar un bloque `finally` tras un bloque `try` simple. En efecto, en ciertos casos, es conveniente no querer interceptar una excepción, pero sí ejecutar un juego de instrucciones necesarias.

La estructura `using` utiliza, en particular, esta técnica: tras la compilación, se transforma en una estructura `try ... finally`. El bloque `finally` contiene una llamada al método `Dispose` de la variable que se utiliza en el bloque `using` inicial.

Consideremos la siguiente sección de código:

```
using (FileStream stream = new FileStream(@"C:\miarchivo.txt",
FileMode.Open))
{
    //Procesamientos
}
```

Tras la compilación, el ensamblado generado contiene un código intermedio equivalente al siguiente código:

```
FileStream stream;
try
{
    stream = new FileStream(@"C:\miarchivo.txt", FileMode.Open);

    //Procesamientos
}
finally
{
    if (stream != null)
        stream.Dispose();
}
```

De este modo, aunque se produzca algún error en la ejecución de los procesamientos, el archivo abierto quedará liberado.

Las herramientas proporcionadas por Visual Studio

Las causas de errores lógicos pueden ser particularmente delicadas de controlar. Visual Studio proporciona, por este motivo, una serie de herramientas que permiten controlar el estado de la ejecución de la aplicación, situar puntos de interrupción o visualizar los datos manipulados por la aplicación en tiempo de ejecución.

1. Control de la ejecución

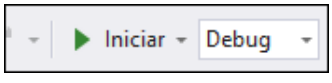
Para poder explorar el código de cara a buscar la causa de un error de ejecución o un error lógico es importante poder controlar la ejecución de una aplicación en condiciones que permitan realizar esta exploración. Visual Studio integra esta posibilidad mediante el uso de un depurador.

En Visual Studio, un proyecto se puede encontrar en tres estados distintos:

- En modo de diseño: el desarrollo está en curso.
- En ejecución: el código se compila y la aplicación se ejecuta y se asocia al depurador integrado.
- En pausa: el depurador ha detenido la ejecución de la aplicación entre dos instrucciones.

a. Arranque

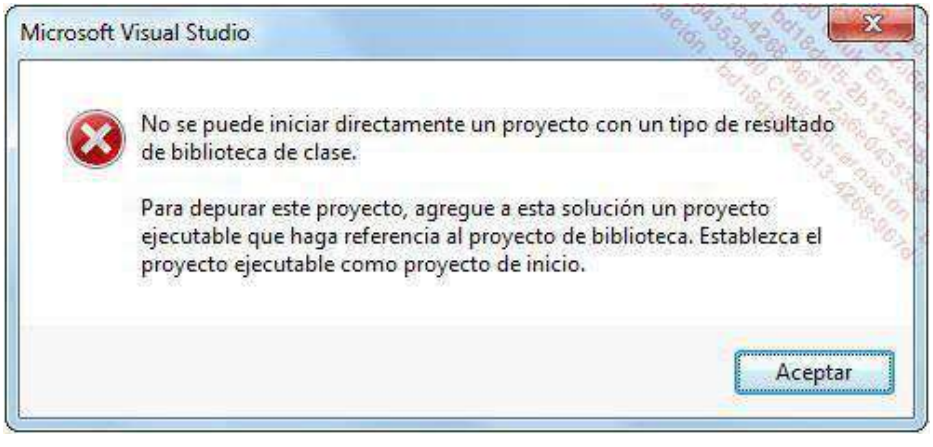
La ejecución de la aplicación en modo de depuración (Debug en Visual Studio) se realiza haciendo clic en el botón **Iniciar** de la barra de herramientas.



También es posible ejecutar la depuración pulsando la tecla [F5] del teclado.

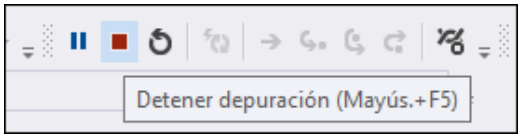
La combinación de las teclas [Ctrl][F5] ejecuta la aplicación en modo normal sin asociarla al depurador de Visual Studio. El proyecto sigue en modo de diseño, pero es imposible compilarlo o ejecutarlo mientras no se cierre la aplicación.

Cuando la solución contiene varios proyectos, el proyecto generado y ejecutado es el que se ha definido como proyecto de inicio. Es necesario, también, que este proyecto genere un archivo ejecutable. En caso contrario, se muestra un error:



b. Detención

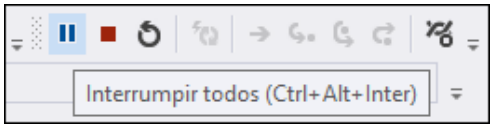
Cuando el proyecto está en modo de ejecución o en pausa es imposible detener completamente su ejecución para pasar de nuevo al modo de diseño. Esta acción se realiza haciendo clic sobre el botón de detención que aparece en la barra de herramientas en modo de ejecución o en pausa.



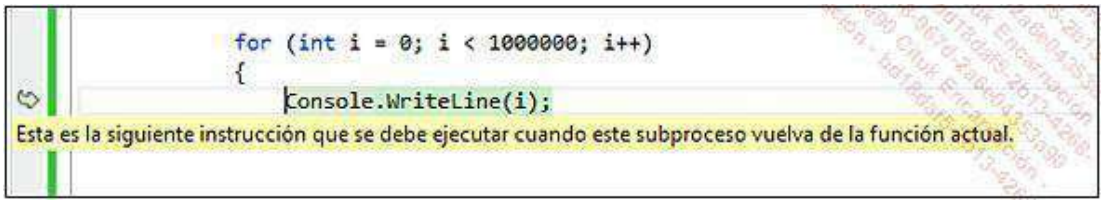
La información que aparece sobre este botón indica que la combinación de teclas [Mayús][F5] detiene también la ejecución de la aplicación en modo de depuración.

c. Pausa

Visual Studio permite poner en pausa la aplicación que está en curso de depuración para permitir inspeccionar el código ejecutado. Esta acción está disponible mediante el botón **Pausa** en la barra de herramientas.



Una vez se pone en pausa la aplicación, se sitúa una marca sobre la siguiente línea que se ejecutará.



Este método es relativamente azaroso puesto que hace falta mucha suerte para detener la aplicación en la línea precisa. Como veremos más adelante, el uso de puntos de interrupción es mucho más eficaz para explorar el código en ejecución.

d. Reanudar

Una vez interrumpida la ejecución de la aplicación, existen varias opciones disponibles. En primer lugar, es posible retomar el curso normal de la ejecución haciendo clic sobre el botón **Continuar** o presionando la tecla [F5]. Es posible, también, seguir en modo paso a paso para seguir los pasos de la ejecución.

Existen tres modos de ejecución paso a paso:

- El paso a paso principal ([F10]),
- El paso a paso detallado ([F11]),
- El paso a paso que sale ([Mayús][F11]).

El paso a paso principal y el paso a paso detallado son casi idénticos. La diferencia entre ambos modos consiste

en la manera de gestionar las llamadas a procedimientos o funciones. Cuando la aplicación se detiene sobre una línea que contiene una ejecución de un método, el paso a paso detallado va a permitirnos entrar en esta llamada y explorar su código. El modo de paso a paso principal ejecutará esta función en un único bloque.

El paso a paso que sale permite ejecutar un método en curso de un bloque y retoma el proyecto en modo pausa en la línea siguiente a la llamada al método.

Una última solución consiste en ejecutar el código en curso hasta la posición definida por el cursor de edición. Para ello, se hace clic en la línea y, a continuación, se utiliza la combinación de teclas [Ctrl][F10]. Esta solución puede resultar muy práctica para salir de la ejecución de un bucle, por ejemplo.

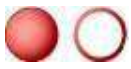
2. Puntos de interrupción

Los puntos de interrupción son herramientas indispensables para depurar una aplicación en C#. Ofrecen la posibilidad al desarrollador de definir la ubicación de una o varias interrupciones en la ejecución del código, lo que permite trabajar sobre escenarios de depuración complejos, que requieren una serie de manipulaciones, pero sin tener que ejecutar la totalidad del código en modo paso a paso.

Los puntos de interrupción pueden ser condicionales, es decir, es posible especificar una condición que debe cumplirse para que el depurador interrumpa la ejecución.

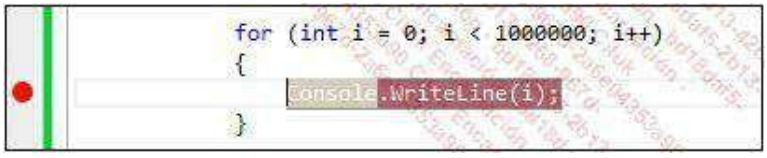
Los TracePoints son muy parecidos a los puntos de interrupción, pero permiten seleccionar una acción a realizar cuando se alcanzan. Esta acción puede ser el paso al modo en pausa y/o mostrar un mensaje que permita trazar la ejecución.

Los puntos de interrupción y TracePoints se representan en Visual Studio mediante iconos que indican su naturaleza y su estado. La siguiente tabla agrupa los distintos iconos. Para cada uno de los cuatro primeros elementos, el icono se divide en dos versiones: la primera se corresponde con el estado activo y la segunda con el estado deshabilitado.

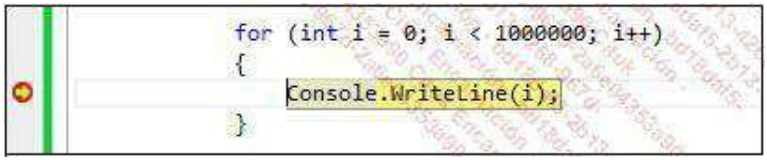
	Representa un punto de interrupción normal.
	Representa un punto de interrupción condicional.
	Representa un TracePoint normal.
	Representa un TracePoint avanzado (con una condición, un contador de pasos o un filtro).
	Punto de interrupción o TracePoint deshabilitado a causa de un error en una condición.
	Punto de interrupción o TracePoint deshabilitado a causa de un problema temporal.

Uso de un punto de interrupción

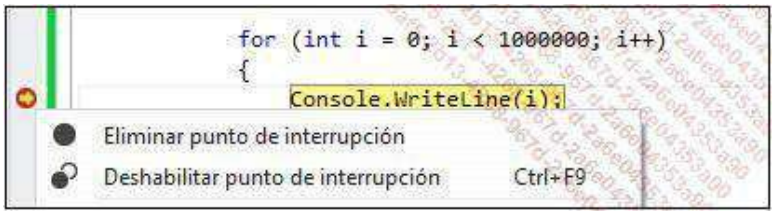
Para situar un punto de interrupción, basta con situar el cursor de edición sobre la línea de código correspondiente y presionar la tecla [F9]. La línea correspondiente se subraya en rojo y se muestra el icono correspondiente al punto de interrupción en el margen de la ventana de edición de código. Es posible, también, hacer clic directamente en el propio margen para definir un punto de interrupción.



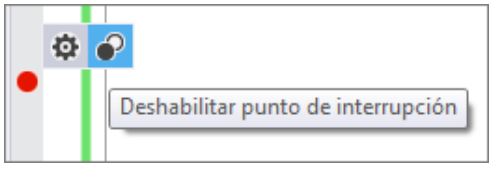
Cuando en la depuración se alcanza un punto de interrupción, Visual Studio pasa a primer plano y la sección de código que contiene el punto de interrupción se muestra en la ventana de edición de código. El color de subrayado de la línea pasa de rojo a amarillo. En este instante, la instrucción todavía no se ha ejecutado y es posible utilizar el modo paso a paso o el paso a paso detallado para visualizar el comportamiento de esta instrucción.



Para deshabilitar el punto de interrupción, haga clic con el botón derecho sobre el icono correspondiente al punto de interrupción para abrir el menú contextual y seleccione, a continuación, **Deshabilitar punto de interrupción**, o bien utilice la combinación de teclas [Ctrl][F9] cuando el cursor de edición esté situado sobre una línea sobre la que se haya definido un punto de interrupción.



También es posible deshabilitarlo utilizando el menú rápido que se muestra cuando se pasa el cursor del ratón por encima del icono que representa el punto de interrupción. El segundo botón de este menú permite habilitar o deshabilitar el punto de interrupción.



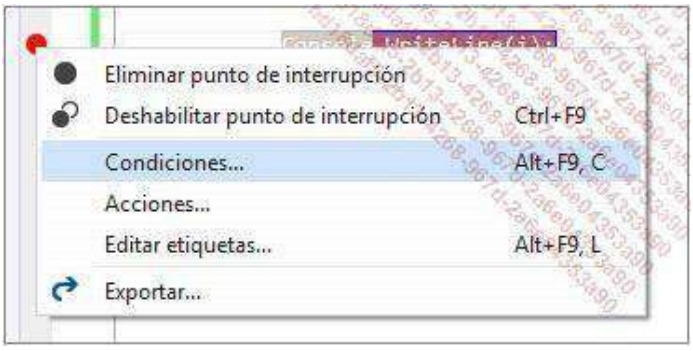
Los puntos de interrupción se eliminan cuando se hace clic en el icono que tienen asociado, o bien mediante la opción **Eliminar punto de interrupción** del menú contextual.

Puntos de interrupción condicionales

Es posible definir condiciones sobre los puntos de interrupción de manera que solamente se interrumpa el flujo de

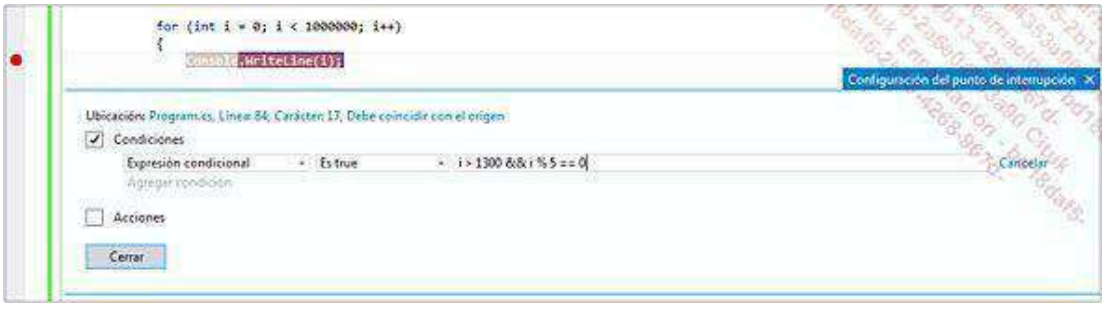
ejecución en aquellos casos particulares en los que, por ejemplo, un contador de bucle alcanza un valor particular.

Para agregar una condición a un punto de interrupción, despliegue el menú contextual haciendo clic sobre el icono y seleccione **Condiciones**.



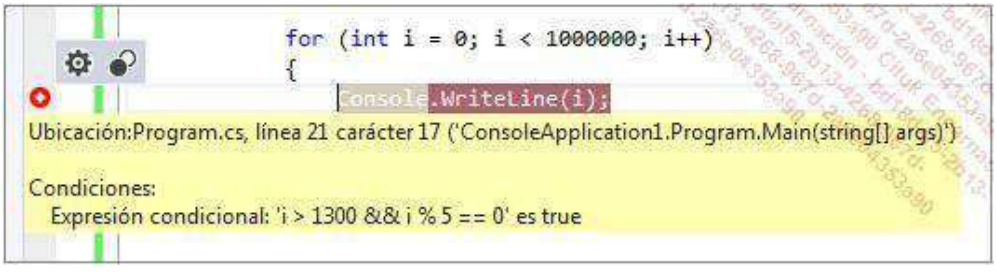
Se abre a continuación la ventana **Configuración del punto de interrupción**, integrada en el editor que permite modificar las condiciones. Es posible introducir una o varias expresiones en C# evaluadas con cada ejecución de la línea de código. Estas se definen mediante tres elementos: un tipo, un modo de evaluación y una expresión.

Por defecto, la expresión esperada es de tipo **Expresión condicional**: se trata de una expresión en C# que devuelve un valor booleano, como una comparación. El modo de evaluación inicial es, por defecto, **Es true**. Esta configuración indica que la ejecución se interrumpe cuando el resultado de la evaluación de la expresión valga `true`.



Aquí, la condición indica que el código debe interrumpirse únicamente cuando la variable i tenga un valor superior a 1300 y este valor sea múltiplo de 5.

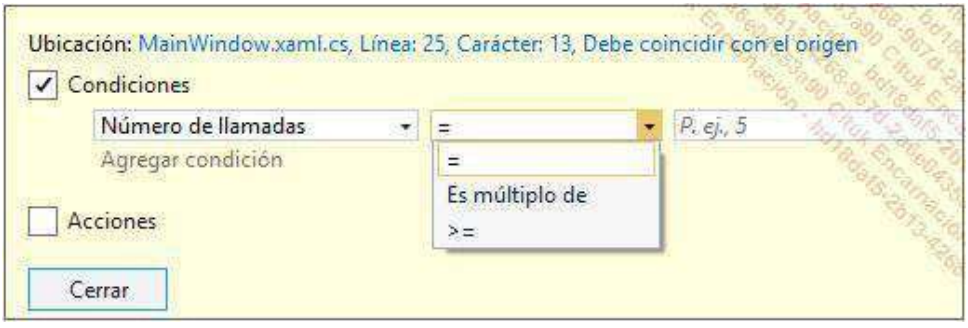
El icono que representa el punto de interrupción cambia de aspecto y la información correspondiente presenta la condición definida.



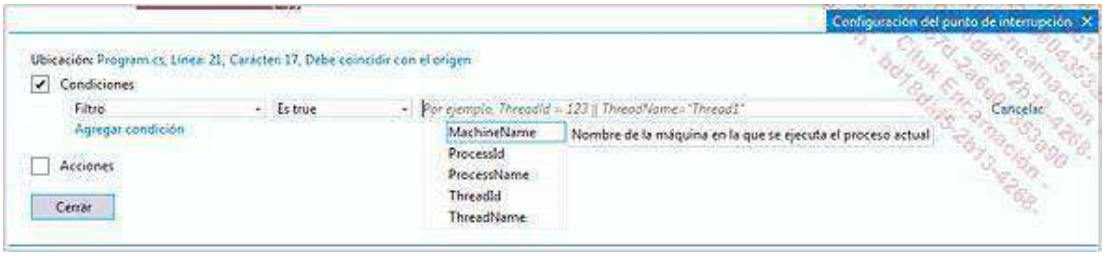
El cambio del modo de evaluación por el valor **cuando cambie** provoca una modificación en el comportamiento del punto de interrupción: la ejecución se interrumpe únicamente cuando el valor de retorno de la evaluación de una condición sea diferente a la evaluación anterior. En el caso descrito anteriormente, la evaluación devuelve `false` hasta que el valor de `i` vale 1304. Cuando pasa de 1305, la evaluación de la condición devuelve `true`: el código se

interrumpe. Pero en la siguiente iteración, `i` vale 1306, de modo que la expresión vale `false`: el punto de interrupción no se produce.

Las condiciones pueden, también, evaluar expresiones relativas al número de accesos realizados a un punto de interrupción. Este se activa solamente cuando se alcanza un cierto número de veces.

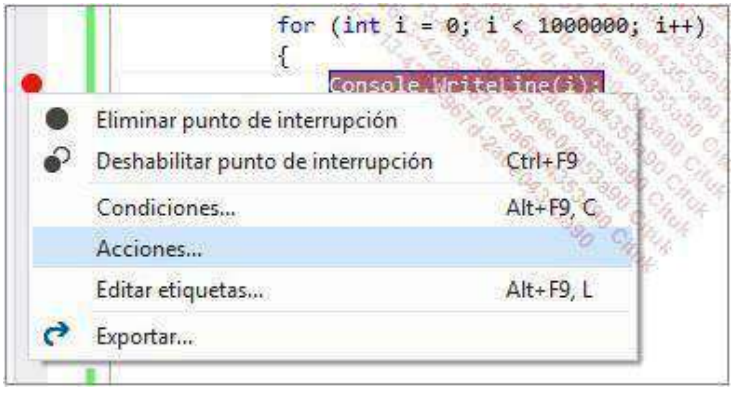


Hay disponible un tercer tipo de expresión: **Filtro**. Este ofrece posibilidades avanzadas de control de los puntos de interrupción. El uso de este tipo de condición permite especificar que el punto de interrupción no se habilite más que para una máquina concreta o un thread en particular.



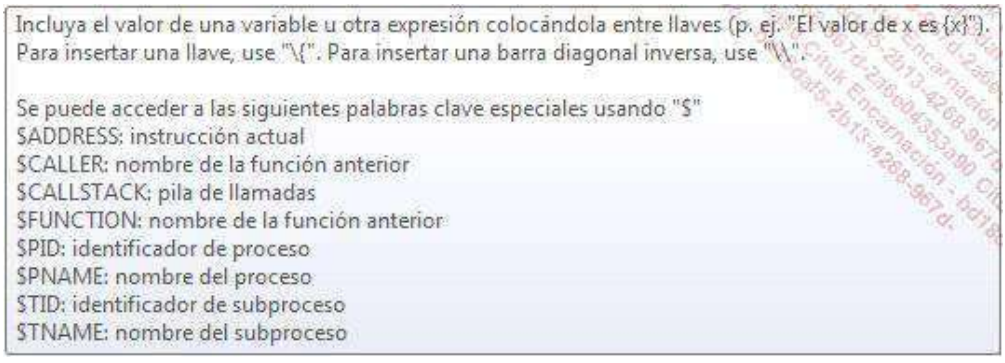
TracePoints

Los TracePoints pueden crearse mediante la opción **Acciones** del menú contextual de un punto de interrupción.



El uso de esta opción abre también la ventana **Configuración del punto de interrupción**. En esta configuración permite definir mediante una macro un mensaje que se mostrará en la ventana de salida de Visual Studio. Para que el punto de interrupción sea realmente un TracePoint es necesario que la opción **Continuar ejecución** esté marcada.

Es posible visualizar las distintas palabras clave que pueden utilizarse en las macros pasando el ratón por encima del icono **Información** situado a la derecha del campo editable.



Si se aplican varias condiciones a un TracePoint, este se activa únicamente si se cumplen dichas condiciones. Conservando la condición creada más arriba, el inicio del resultado en la ventana de salida es el siguiente:

```
Function: ConsoleApplication2.Program.Main(string[]), El valor
de i es 1305
Function: ConsoleApplication2.Program.Main(string[]), El valor
de i es 1310
Function: ConsoleApplication2.Program.Main(string[]), El valor
de i es 1315
Function: ConsoleApplication2.Program.Main(string[]), El valor
de i es 1320
```

3. Visualizar el contenido de las variables

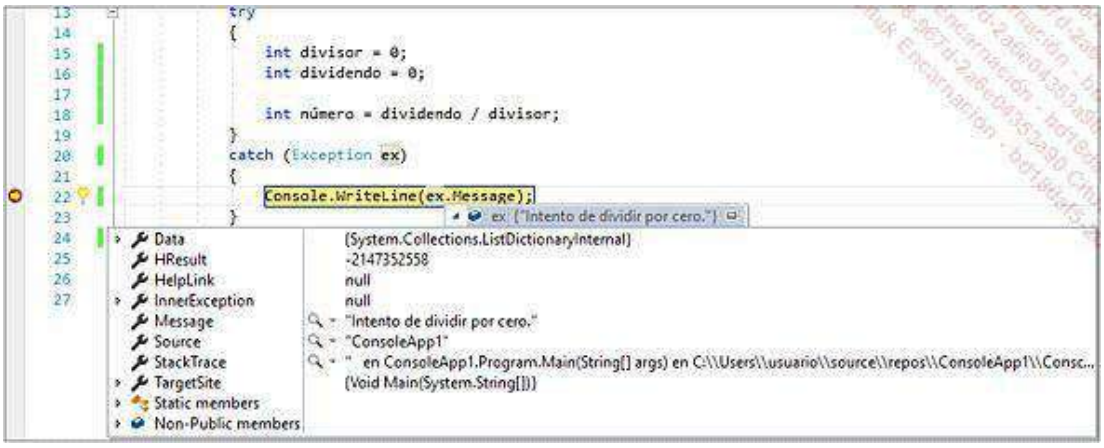
El depurador permite al desarrollador seguir el funcionamiento de su aplicación, instrucción a instrucción si es necesario. Es importante, también, que pueda inspeccionar el resultado de una operación o el valor de los parámetros que se pasan a un método para poder comprender el motivo de un resultado inesperado o de un error de ejecución.

Visual Studio provee diversas herramientas cuando el proyecto se encuentra en modo pausa para acceder a estos valores.

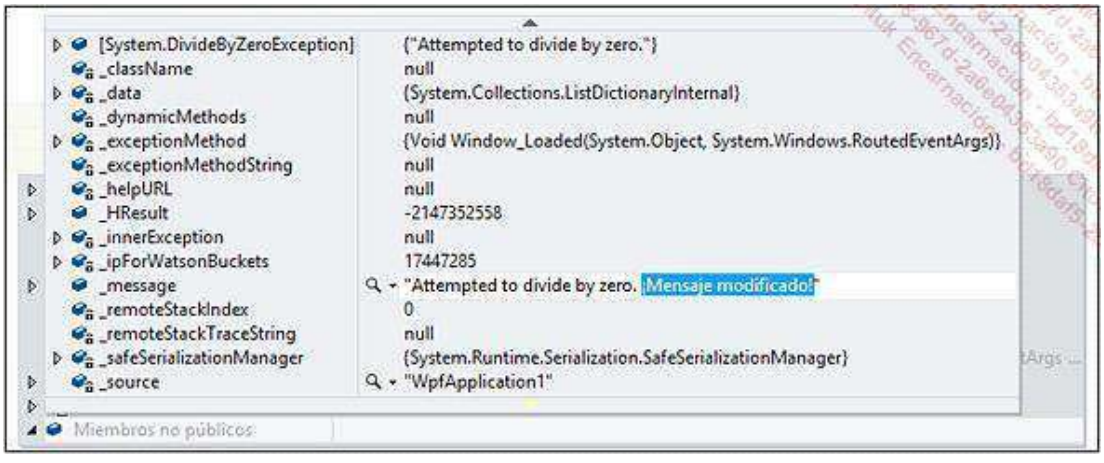
a. DataTips

Los DataTips constituyen el primer medio de acceso al contenido de las variables, dada su simplicidad y su accesibilidad. Basta con situar el cursor del ratón sobre el nombre de la variable, accesible por el código sobre el que se ha detenido el depurador para ver cómo aparece una pequeña ventana que contiene el valor de la variable.

Si la variable es de un tipo complejo, esta ventana proporciona un botón con forma de triángulo que permite inspeccionar sus propiedades.



También es posible modificar los valores de las variables que se muestran, lo cual puede resultar muy práctico para validar un comportamiento.



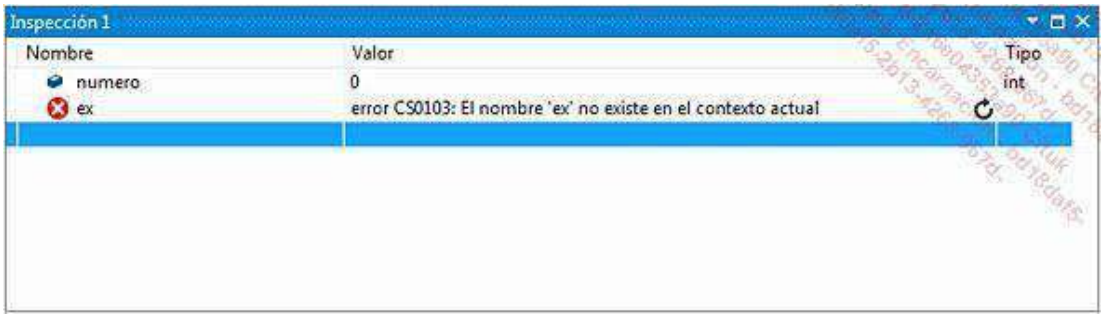
Los DataTips desaparecen automáticamente cuando el cursor del ratón sale de su superficie.

b. Ventanas de inspección

Las ventanas de inspección 1 a 4 están accesibles mediante el menú **Depurar - Ventanas - Inspección** de Visual Studio. Estas ventanas tienen como objetivo permitir la visualización rápida y simultánea de valores de distintas variables.

Para agregar una ventana de inspección a una variable es preciso hacer clic con el botón derecho sobre el nombre de la variable y seleccionar la opción **Agregar inspección**. Es posible, también, agregar una variable o expresión declarándola en la columna **Nombre**.

Los valores de las distintas expresiones se evalúan a lo largo del proceso de depuración de la aplicación. Si alguna de ellas no pudiera evaluarse, el icono que precede a la excepción mostraría un error y la columna **Valor** mostrará un mensaje indicando que la expresión no existe en el contexto actual.



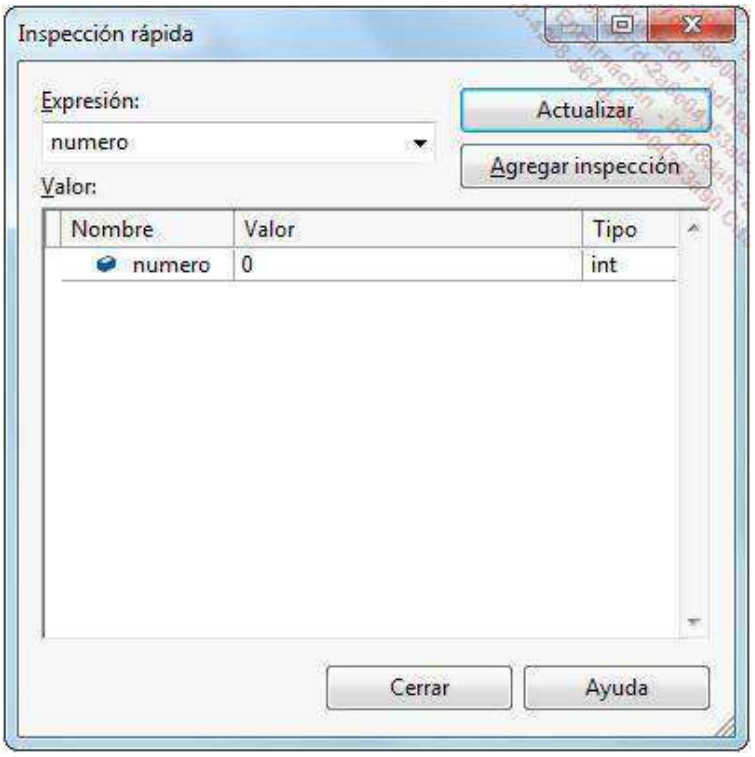
Como en un DataTip, es posible modificar el valor de una variable en una ventana de inspección.

Es posible eliminar una inspección mediante la opción **Eliminar inspección** haciendo clic con el botón derecho sobre el elemento a eliminar o utilizando la tecla [Supr].

c. Ventana de inspección rápida

La ventana de inspección rápida utiliza el mismo principio de funcionamiento que las ventanas de inspección. No permite, en cambio, visualizar más que un valor cada vez, lo que limita bastante su interés respecto a la ventana de inspección. Además, esta ventana es modal, lo que implica que la ventana debe cerrarse para poder seguir con la depuración.

Está accesible mediante el menú **Depurar - Inspección rápida**, el atajo de teclado [Ctrl] D, Q o mediante la opción **Inspección rápida**, haciendo clic con el botón derecho sobre el nombre de la variable.



d. Ventana Automático

La ventana **Automático** presenta una visión rápida de los valores de las variables utilizadas en la instrucción en curso y la instrucción anterior. Es muy similar a la ventana de inspección, pero no permite agregar valores para visualizarlos.

Está accesible mediante la opción de menú **Depurar - Automático** o mediante el atajo de teclado [Ctrl][Alt] V, A.

Como en una ventana de inspección, es posible modificar el valor de una variable haciendo doble clic en la columna **Valor**.

e. Ventana de variables locales

Esta última ventana es idéntica a la ventana **Automático**, pero las variables de las que se muestra el valor son variables locales al procedimiento o a la función en curso de ejecución.

Está accesible mediante el menú **Depurar - Ventanas - Variables locales** o mediante el atajo de teclado [Ctrl] D, L.

4. Compilación condicional

Si bien no está dedicada exclusivamente a la depuración, la compilación condicional es una herramienta muy interesante para realizar el seguimiento y diagnosticar los problemas, en particular en fase de desarrollo.

El principio de la compilación condicional es muy simple: se integra o no una sección de código en el ensamblado generado en función de una condición predefinida.

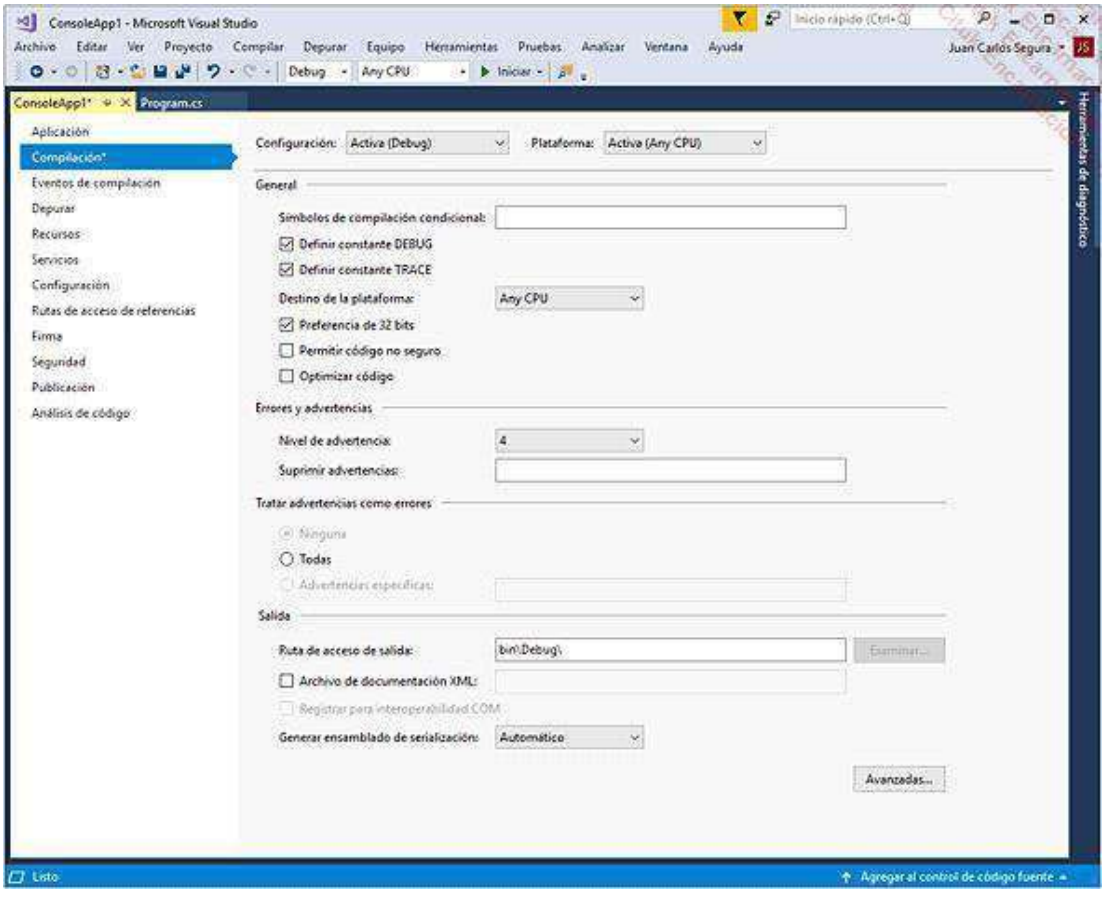
Esta condición se basa, obligatoriamente, en la existencia o no de una o varias constantes de compilación.

Las constantes de compilación no poseen valor: existen o no. Es posible definirlas de dos maneras distintas: de manera global, a nivel de un proyecto, o de manera local, en el código.

Constantes de compilación a nivel de un proyecto

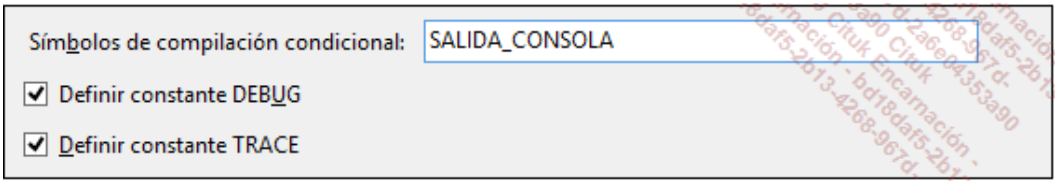
Para definir de manera visual una constante de compilación a nivel de un proyecto es preciso acceder a la ventana de propiedades del proyecto.

- En el explorador de soluciones, haga clic con el botón derecho sobre el proyecto correspondiente y seleccione el menú **Propiedades**.
- Una vez abierta la ventana de propiedades, seleccione el menú **Compilar**.



La primera parte de la sección **General** permite definir constantes personalizadas, así como las constantes `DEBUG` y `TRACE`, que utilizan ciertas clases del framework .NET.

Cuando se necesita definir varias constantes de compilación, estas deben estar separadas por el símbolo ; (punto y coma).



Definir la constante `SALIDA_CONSOLA` tiene como efecto aplicar el parámetro `/define:SALIDA_CONSOLA` al proyecto en tiempo de compilación.

Constantes de compilación locales

Es posible definir una constante de compilación para un único archivo en el código.

Para ello, es necesario utilizar la directiva preprocesador `#define` seguida del nombre de la constante a definir. Un archivo puede contener varias directivas `#define`, pero deben situarse todas al principio del archivo, antes de las demás instrucciones.

```
#define SALIDA_CONSOLA
```

En ocasiones puede ser necesario definir una constante de compilación únicamente sobre una sección de un archivo. En este caso, se utiliza la directiva `#undef` en la sección en la ya no se desea que esté definida la constante.

Implementar la compilación condicional

La compilación condicional permite incluir código en función de la existencia o no de una constante de compilación. Para implementarla, es necesario poder verificar esta existencia.

Esta verificación se realiza mediante la directiva de precompilación `#if ... #endif`. Esta instrucción es similar a la instrucción `if`: evalúa una condición e incluye el código que contiene si la evaluación devuelve `true`.

Para incluir código en función de la existencia de la constante `SALIDA_CONSOLA` definida más arriba, escribiremos:

```
#if SALIDA_CONSOLA
   Console.WriteLine("SALIDA_CONSOLA está definida");
#endif
```

En el marco de la depuración de una aplicación también puede resultar interesante definir instrucciones `#if` en ciertos lugares para registrar la máxima información posible, en particular en el interior de bloques `catch { }`.

Presentación de WPF

Las aplicaciones Windows son aplicaciones de ventanas que presentan datos de una forma más gráfica y agradable que la ventana de línea de comandos. Para desarrollar este tipo de aplicaciones, .NET proporciona desde sus inicios la tecnología **Windows Forms**. La versión 3.0 del framework .NET, aparecida en 2006, llegó con una nueva tecnología de presentación: **WPF** (*Windows Presentation Foundation*). Aporta algunas mejoras notables tanto a nivel técnico como en lo relativo al desarrollo:

- La definición de la interfaz se realiza mediante el lenguaje de etiquetas **XAML** (*eXtensible Application Markup Language*) y no mediante código C#.
- Se **desacopla** la interfaz gráfica del código de negocio fácilmente gracias a la noción de **binding**.
- La **visualización** ya no se basa en el componente de software GDI sino en **DirectX**, lo que implica que algunos cálculos puede realizarlos la **GPU**.
- Todos los componentes de WPF utilizan el **diseño vectorial**.
- En un componente, el aspecto gráfico y el aspecto funcional están muy poco ligados, lo que permite modificar la parte gráfica de los componentes sin afectar a su comportamiento.

A pesar de lo robusta que es, la tecnología Windows Forms se deja a un lado con frecuencia en beneficio de WPF. Los motivos principales de esta elección están ligados por un lado a la experiencia del usuario, dado que **WPF** permite crear de manera sencilla **aplicaciones fluidas** con un **aspecto moderno**, y por otro lado a motivos técnicos, pues las posibilidades de desacoplo que ofrece WPF mejoran las **pruebas** y la **mantenibilidad** de las aplicaciones.

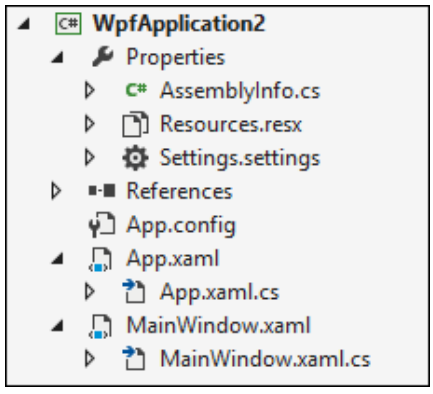
Microsoft trabaja, actualmente, sobre WPF y sobre tecnologías que utilizan XAML. Un equipo de desarrolladores del gigante de software ha comentado, durante la conferencia BUILD 2014 que la tecnología **Windows Forms** se encuentra, actualmente, en **periodo de mantenimiento**, lo que significa que ya no aparecerán actualizaciones funcionales, si bien los problemas detectados sí se irán corrigiendo.

1. Estructura de una aplicación WPF

En el momento de su creación, un proyecto WPF está compuesto por varios archivos:

- Los archivos `App.xaml` y `App.xaml.cs` contienen la declaración y la implementación de la clase principal de la aplicación: de hecho, contiene el punto de entrada al programa. El archivo `.xaml` contiene, a su vez, todos los recursos necesarios para el correcto funcionamiento de la aplicación, mientras que el archivo de code-behind asociado contiene los controladores de eventos relativos al ciclo de vida de la aplicación o las secciones de código necesarias para su inicio.
- Los archivos `MainWindow.xaml` y el archivo de code-behind `Main Windows.xaml.cs` contienen, respectivamente, la descripción de la ventana principal de la aplicación y el código C# asociado.
- El archivo `app.config` contiene los parámetros de configuración de la aplicación.
- `AssemblyInfo.cs` contiene los metadatos que describen el ensamblado resultante de la compilación del proyecto: título, versión, información de copyright... Esta información se describe en forma de atributos.
- El archivo `Resources.resx` contiene, generalmente, cadenas de caracteres que pueden utilizarse en varios lugares dentro de la aplicación, y es capaz, también, de encapsular recursos de tipo binario.
- El archivo `Settings.settings` permite almacenar parámetros vinculados a la aplicación o a un usuario de la aplicación: tamaño de la letra, colores o situación de las barras de herramientas, por ejemplo.

Estos distintos archivos se organizan de la siguiente manera:



2. XAML

La tecnología WPF utiliza el lenguaje XAML para la **creación de interfaces gráficas**. Este lenguaje es un **dialecto de XML** que permite instanciar objetos .NET de manera declarativa, es decir, mediante el uso de etiquetas, de manera similar a como lo hace HTML. Los distintos objetos están anidados a partir de un elemento raíz, formando un árbol lógico.

La definición de un botón mediante código XAML tiene el siguiente aspecto:

```
<Button Height="30" Width="120">Esto es un botón</Button>
```

La etiqueta `<Button>` instancia un objeto de tipo `Button`, mientras que los atributos XML asignan valor a las propiedades `Height` y `Width` del botón. Como cualquier elemento XML, una etiqueta XAML puede contener otras etiquetas o un valor textual.

Es importante saber que todo lo que puede codificarse mediante XAML puede codificarse también con C#. El

equivalente imperativo del código XAML anterior es el siguiente:

```
Button botón = new Button();
botón.Height = 30;
botón.Width = 120;
botón.Content = "Esto es un botón";
```

Vemos una nueva propiedad `Content` para la que no se proporcionaba ningún valor, de manera explícita, en XAML. Esta propiedad es, de hecho, la **propiedad por defecto** para el tipo `Button`, lo que significa que el valor textual o XAML situado en el interior de la etiqueta `Button` asigna el valor a la propiedad `Content` del objeto `Button`.

La propiedad por defecto de un tipo se establece definiendo en el tipo un atributo `ContentPropertyAttribute` especificando el nombre de la propiedad.

La **sintaxis elemento-propiedad** permite asignar un contenido XAML complejo a una propiedad. Autoriza el uso de una propiedad como elemento XML anidado. El nombre del elemento debe mostrar el tipo sobre el que se opera, seguido de un punto y, a continuación, el nombre de la propiedad correspondiente. Según esta sintaxis, el código XAML de creación de un botón que hemos visto antes puede escribirse de la siguiente manera:

```
<Button Height="30" Width="120">
    <Button.Content>Esto es un botón</Button.Content>
</Button>
```

En cambio, no todo el código C# tiene su equivalente en XAML, pues es imposible, en particular, invocar a un método directamente de manera declarativa. Si bien XAML es muy potente gracias a la gran cantidad de objetos que permite utilizar, los escenarios avanzados requieren, a menudo, el uso de código .NET.

Si bien no se presentó inicialmente con WPF, el lenguaje de descripción XAML se utiliza, actualmente, para el desarrollo de aplicaciones que no utilizan la tecnología WPF, como por ejemplo aplicaciones para Windows Store o Windows Phone.

a. Plantillas

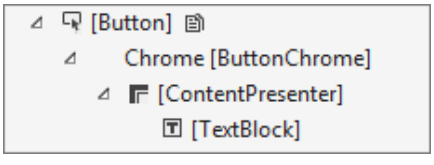
Los objetos se definen bajo la forma de un grafo que compone el árbol lógico de una pantalla. Este árbol es distinto del árbol visual, que es el resultado de la transformación del árbol lógico tras la aplicación de distintas *plantillas* sobre cada control.

La parte visual de un control gráfico se define en XAML mediante la creación de un objeto de tipo `ControlTemplate`. Se trata de una plantilla de diseño que representa el resultado deseado mediante un árbol formado por objetos más simples. Se aplica al control tras su carga.

El árbol lógico siguiente es extremadamente sencillo: contiene una única etiqueta.

```
<Button Height="30" Width="120" Content="Esto es un botón" />
```

El árbol visual correspondiente es algo más complejo:



Esta representación del árbol visual de un botón se obtiene mediante la ventana **Árbol visual dinámico** de Visual Studio. Esta puede abrirse desde la depuración de una aplicación WPF en el menú **Depurar - Ventanas - Árbol visual dinámico**.

A continuación se agregan dos elementos: un objeto de tipo `Border` y un objeto de tipo `ContentPresenter`. Forman parte del `ControlTemplate` asociado a la clase `Button`. El objeto de tipo `TextBlock`, agregado por el `ContentPresenter`, ha encapsulado nuestro texto en su propiedad `Text`.

b. Espacios de nombres

Como ocurre en el código C#, es necesario importar los espacios de nombres en el código XAML para poder utilizar los tipos que contienen. Para realizar esta operación no es posible, evidentemente, utilizar una declaración `using` de C#, sino las declaraciones de espacios de nombres de XML.

Estas declaraciones son atributos XML que respetan una sintaxis particular:

```
xmlns:<prefijo>="clr-namespace:<espacio de nombres>"
```

En cada ventana WPF son obligatorios dos espacios de nombres, y deben estar presentes en el elemento raíz. El primero importa el framework WPF y su implementación de XAML, mientras que el segundo importa varios elementos globales de XAML.

Estas dos declaraciones permiten utilizar las bases de WPF en el conjunto del documento en edición.

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Cuando se crea una ventana mediante la plantilla presente en Visual Studio, su declaración ya incluye estos espacios de nombres de manera predeterminada.

Afortunadamente, importar espacios de nombres .NET es bastante más sencillo. Para el espacio de nombres `Ejemplo.Controles`, la declaración se haría de la siguiente manera:

```
xmlns:controlesEjemplo="clr-namespace:Ejemplo.Controles"
```

El control `Calendario` incluido en este espacio de nombres puede utilizarse, a continuación, prefijando su nombre de tipo por el alias del espacio de nombres `controlesEjemplo` seguido del símbolo `:`, lo que produce la siguiente instanciación:

```
<controlesEjemplo:Calendario />
```

3. Contexto de datos y binding

El concepto de **binding** es, posiblemente, el más importante para realizar un uso óptimo de las capacidades de WPF. Se corresponde con un **vínculo unidireccional o bidireccional** entre un elemento XAML y un **contexto de datos** (`DataContext`) y permite propagar datos del código C# hacia el código XAML (y/o a la inversa) sin que el desarrollador tenga que realizar una transferencia de información mediante código C#. Este mecanismo es la clave del **desacople** entre el código de negocio y la interfaz gráfica.

Se define un vínculo de datos en el código XAML mediante una expresión delimitada por llaves y cuyo primer término es la palabra clave `Binding`. Este vínculo puede estar seguido de una lista de parámetros que definen el origen de datos, la ruta de la propiedad con la que se debe establecer el vínculo o incluso el modo de enlace.

Por defecto, un binding transmite el resultado de la llamada al método `ToString` del objeto. Para objetos complejos, el resultado que se muestra, si no se ha redefinido el método, es el nombre completo de su tipo.

- Cree un nuevo proyecto WPF en Visual Studio mediante el cuadro de diálogo **Nuevo proyecto** (menú **Archivo - Nuevo - Proyecto** o [Ctrl][Mayús] N) seleccionando la opción **Aplicación WPF** y llámelo **PruebasBinding**.
- Edite el contenido del archivo `MainWindow.xaml` del proyecto que acaba de crear de modo que contenga el siguiente código:

```
<Window x:Class="PruebasBinding.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <TextBox Height="30" Width="120" Text="{Binding Path=Nombre,
Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />
    </Grid>
</Window>
```

Este código crea una zona para introducir texto en una ventana y asocia a su propiedad `Text` un vínculo bidireccional (`Mode=TwoWay`) hacia la propiedad Nombre del contexto de datos (`Path=Nombre`). La transferencia de información del control hacia el contexto de datos se realizará cada vez que la propiedad sobre la que se haya definido el binding sufra alguna modificación (`UpdateSourceTrigger=PropertyChanged`).

- Ejecute el proyecto presionando la tecla [F5] o el botón **Iniciar** de la barra de herramientas de Visual Studio (habiendo definido previamente el proyecto como proyecto de inicio).

En este punto, no se produce nada especial. En efecto, no se ha definido ningún contexto de datos y, por tanto, no existe ninguna fuente de datos especificada explícitamente. Para definir el contexto de datos de la ventana es preciso asignar un valor a su propiedad `DataContext`.

- Edite el constructor de la ventana, en `MainWindow.xaml.cs`, para definir el contexto de datos de la ventana como la propia ventana.

```
public MainWindow()
{
   this.InitializeComponent();
   this.DataContext = this;
}
```

Si arranca la aplicación, la interfaz se comportará exactamente de la misma manera. En cambio, la ventana **Salida** de Visual Studio presenta una nueva línea de información:

```
System.Windows.Data Error: 40 : BindingExpression path
error: 'Nombre' property not found on 'object' ''MainWindow'
(Name='')'. BindingExpression:Path=Nombre;
DataItem='MainWindow' (Name=''); target element is 'TextBox'
(Name=''); target property is 'Text' (type 'String')
```

Este mensaje indica que se ha producido un error en la creación del vínculo de datos, pues la propiedad `Nombre` no se encuentra en el objeto `MainWindow` que se utiliza como contexto de datos. Provee, también, información que permite identificar el objeto XAML y la propiedad sobre la que se ha situado el vínculo erróneo.

- Agregue la definición de la propiedad siguiente en la clase `MainWindow` de cara a corregir el error.

```
private string nombre;
public string Nombre
{
    get { return nombre; }
```

```
    set { nombre = value; }
}
```

- Para visualizar el correcto funcionamiento del vínculo, agregue una asignación del valor a la propiedad `Nombre` en el constructor de la clase.

```
public MainWindow()
{
    InitializeComponent();
    Nombre = "Sebastián";
    this.DataContext = this;
}
```

Con esta configuración, la aplicación muestra una zona de texto en la que se presenta el valor de la propiedad `Nombre`, pero las modificaciones realizadas sobre este valor tras un clic en un botón, por ejemplo, no se reflejarán en el control `TextBox` a pesar de la existencia de un vínculo de datos.

- Para comprobarlo, agregue la siguiente línea justo a continuación de la declaración del control `TextBox` en `MainWindow.xaml`:

```
<Button Height="30" Width="120" VerticalAlignment="Top"
Click="Button_Click" Content="Modificar el contenido del área de
texto" />
```

- Agregue, a su vez, el controlador de eventos siguiente a la clase `MainWindow`:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Nombre = "Nombre modificado";
}
```

La no propagación de la modificación es, de hecho, normal. La tecnología WPF utiliza el evento `PropertyChanged` de la interfaz `INotifyPropertyChanged` para informar a los elementos gráficos acerca de las modificaciones de valor en sus propiedades. Es necesario, a continuación, que el objeto que se utiliza como origen de datos (aquí, la clase `MainWindow`) implemente esta interfaz.

A continuación se muestra el código completo de la clase `MainWindow` una vez implementada esta interfaz:

```
public partial class MainWindow : Window, INotifyPropertyChanged
{
    private string nombre;
    public string Nombre
    {
        get { return nombre; }
        set
        {
            nombre = value;
            OnPropertyChanged("Nombre");
        }
    }
```

```
    public MainWindow()
    {
        InitializeComponent();
        this.DataContext = this;
        Nombre = "Sebastián";
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        this.Nombre = "Nombre modificado";
    }

    private void OnPropertyChanged(string nombrePropiedad)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new
PropertyChangedEventArgs(nombrePropiedad));
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

Uso de controles

La sección anterior nos ha mostrado, brevemente, la estructura de una aplicación WPF. Habrá podido observar que la **interfaz gráfica está compuesta por controles** definidos en el **código XAML** o, con menor frecuencia, en C#.

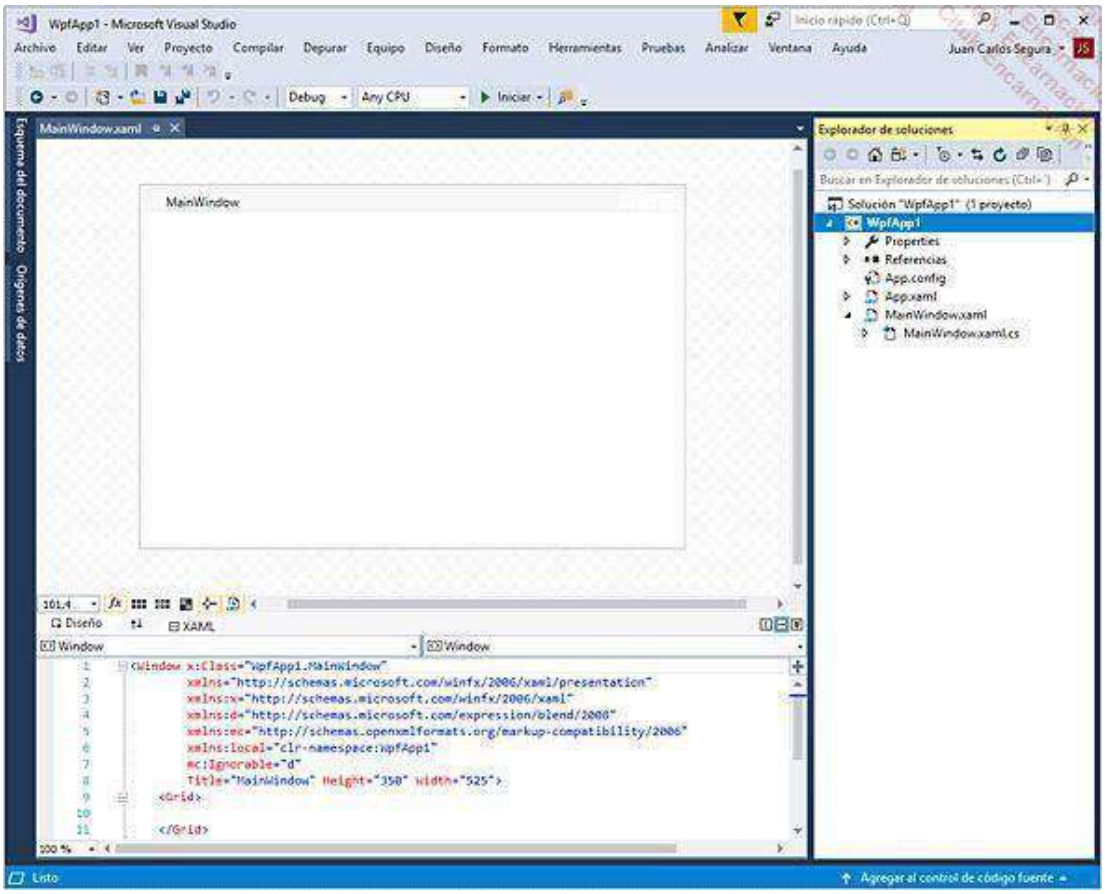
Estos controles forman parte de una jerarquía que permite tener propiedades comunes, la mayor parte del tiempo ligadas a su posicionamiento. Cada control hereda, de este modo, de un control padre al que se le agregan funcionalidades de cara a proveer un nuevo conjunto formado por un elemento visual y un comportamiento funcional asociado.

Visual Studio integra, de manera nativa, un **diseñador visual** que permite manipular controles de manera gráfica, y por tanto más simple. Genera, también, código XAML que define la interfaz. A diferencia del diseñador visual de Windows Forms, el diseñador de WPF crea código XAML que es posible manipular directamente, donde cualquier modificación sobre el código tiene su reflejo en la parte de diseño gráfico, y a la inversa.

1. Agregar controles

Tras la creación de un nuevo proyecto de aplicación WPF, Visual Studio muestra, por defecto, varios elementos:

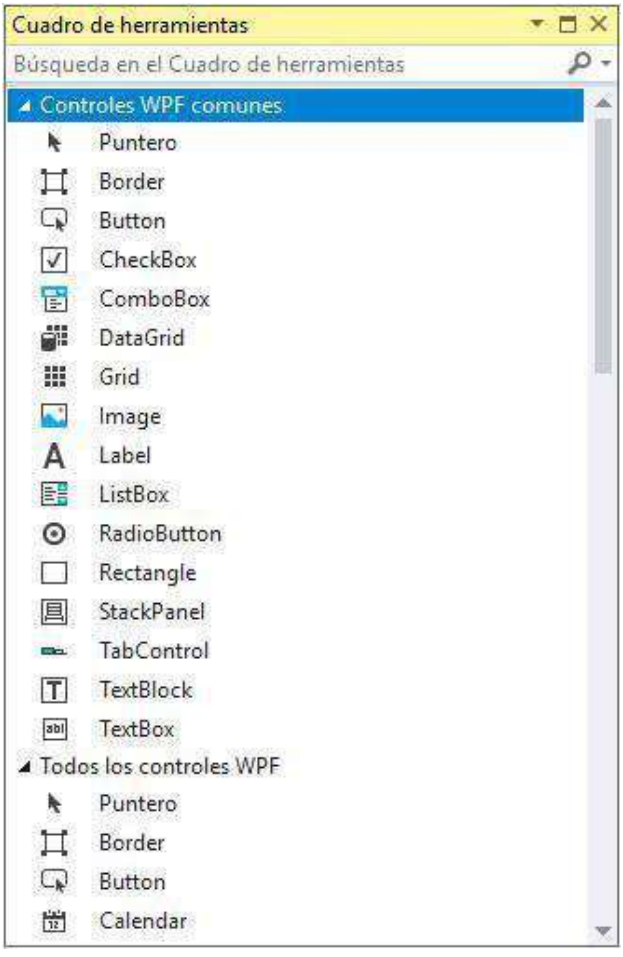
- El **Explorador de soluciones**,
- La ventana **Propiedades**,
- Una **ventana de edición** dividida en dos secciones: el diseñador visual en la zona superior, y el editor de código XAML situado en la zona inferior.



La ventana de edición está destinada a la modificación de la ventana principal de la aplicación, llamada

`MainWindow`. El diseñador visual presenta una vista de esta ventana, que no tiene de momento ningún contenido.

Varias ventanas, ocultas automáticamente, se posicionan en la zona izquierda del entorno. En el contexto de la creación o modificación de una interfaz gráfica, la más importante es la ventana **Cuadro de herramientas**.

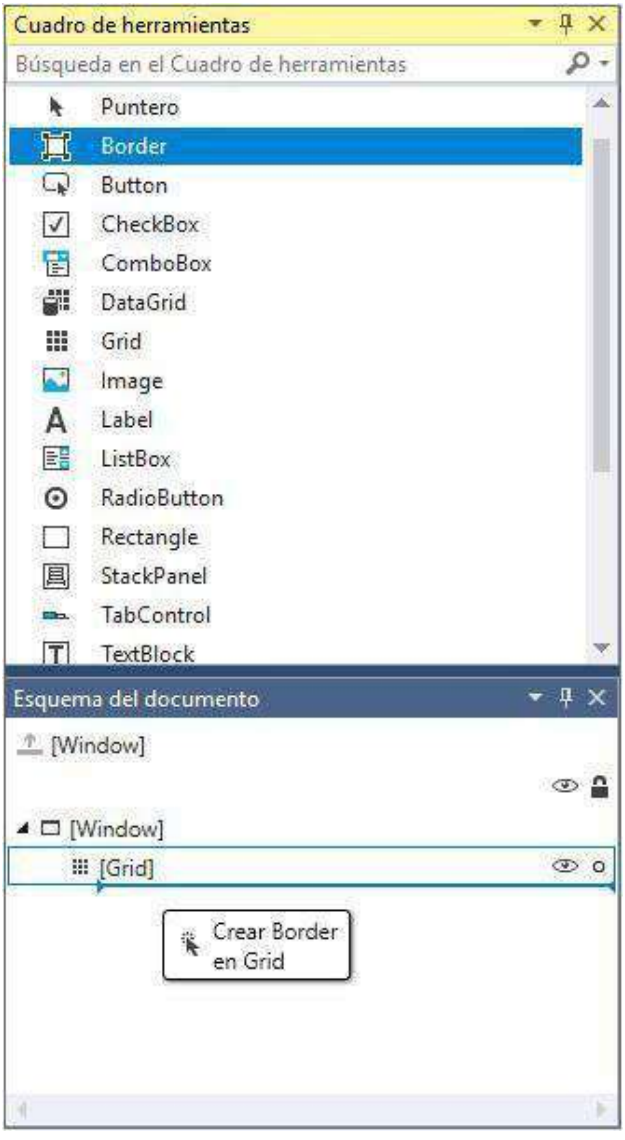


Esta ventana enumera distintos controles que pueden utilizarse en una interfaz WPF. Permite, también, seleccionarlos y situarlos con ayuda del ratón. Este enfoque resulta mucho más rápido que la edición del código fuente XAML.

Es posible utilizar tres soluciones para situar controles en una ventana WPF:

- Haciendo **doble clic** sobre un control en el cuadro de herramientas se sitúa un ejemplar del control en la ventana. El control se sitúa, por defecto, en la esquina superior izquierda de la ventana. Sus propiedades de dimensionamiento son las especificadas por defecto.
- El uso de la acción **arrastrar y colocar** sobre un elemento del cuadro de herramientas hacia el diseñador visual permite situar un control en el lugar deseado. Sus dimensiones son las definidas por defecto.
- La selección de un elemento en el cuadro de herramientas permite **diseñar una zona rectangular** cuyo tamaño y posición los define el usuario. El control se situará en esta zona.

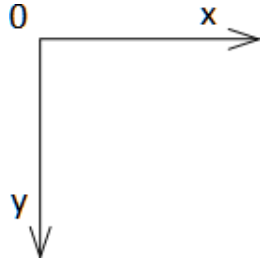
La ventana **Esquema del documento** ofrece una visualización rápida del árbol de la ventana en edición. Permite, también, agregar un control en una ubicación precisa del árbol. Puede utilizar la acción arrastrar y colocar sobre un elemento del cuadro de herramientas hacia esta ventana. Cuando se pasa por encima de esta ventana, aparecen algunas indicaciones que le ayudan a situar el control en la ubicación lógica deseada.



El control creado en la arborescencia se ubica, por defecto, en su control contenedor y sus propiedades de dimensionamiento son las definidas por defecto por el diseñador visual. Este modo de creación del control es particularmente útil cuando la interfaz es compleja.

2. Posición y dimensionamiento de controles

Los controles se posicionan, por defecto, en las coordenadas 0;0 de su contendor (cuando éste lo permite) y se corresponden con la esquina superior izquierda del control padre. WPF utiliza, en efecto, un sistema de coordenadas cartesianas en las que el eje de ordenadas está orientado hacia abajo. La siguiente figura ofrece una representación visual de este sistema.



Evidentemente, la posición por defecto es en pocas ocasiones la deseada por el usuario. Los controles WPF poseen una gran cantidad de propiedades que permiten modificar este comportamiento, así como su dimensionamiento.

Height y Width

Las propiedades `Height` y `Width` permiten definir de manera absoluta o relativa la altura y anchura de un control. Pueden contener tres tipos de valores para cumplir su objetivo:

- Un **valor numérico absoluto** en píxeles.
- Un **valor numérico** absoluto seguido de una **unidad de medida**. Esta unidad puede ser `px` (píxeles), `in` (pulgadas), `cm` (centímetros), `pt` (puntos - unidad que se utiliza en tipografía).
- El valor `Auto`, que indica que el control debe **dimensionarse en base a sus necesidades y dentro de los límites de su control contenedor**. Algunos controles utilizarán, por defecto, todo el espacio disponible, mientras que otros adaptarán su tamaño en función de su contenido.

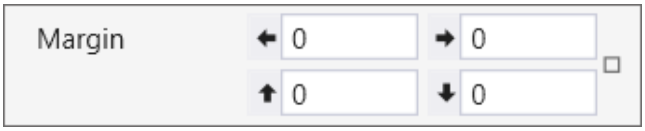
HorizontalAlignment y VerticalAlignment

La posición de un control en su control padre puede gestionarse de manera automática en función de una alineación. Las propiedades `HorizontalAlignment` y `VerticalAlignment` permiten implementar este tipo de posicionamiento proporcionando un valor de enumeración que indica la alineación que se desea utilizar: `Left`, `Center`, `Right` o `Stretch` horizontalmente y `Bottom`, `Center`, `Top` o `Stretch` verticalmente.

Los tres primeros valores son relativamente explícitos. Permiten alinear el control en el borde izquierdo, en el centro, o en el borde derecho de su control padre. El valor `Stretch` permite imponer al control que utilice todo el espacio disponible horizontal o verticalmente.

Margin

La propiedad `Margin` de un control define una zona vacía alrededor de él que permite **desplazarlo respecto a su posición normal**. Existen cuatro valores asociados a esta propiedad que corresponden a los cuatro bordes del control. La siguiente figura muestra la definición de estos valores en la ventana de **Propiedades** de Visual Studio.



Cuando se modifican estos valores directamente en el código XAML deben declararse en el siguiente orden: **izquierda**, **arriba**, **derecha**, **abajo**.

```
<Button Margin="0 10 0 0" />
```

En este caso, el control `Button` se desplazará 10 píxeles hacia abajo respecto a su posición normal en el contenedor.

Los valores definidos para esta propiedad pueden, como las propiedades `Width` y `Height`, especificar una unidad de medida particular. Visual Studio no soporta unidades de medida en la ventana de propiedades, por lo que es obligatorio, si desea utilizarlas, definirlas en el editor de código fuente XAML.

```
<Button Margin="1cm 2px 3in 4pt"/>
```

La propiedad `Margin` se utiliza en el diseñador visual de Visual Studio para posicionar un control en un contenedor de tipo `Grid`, por ejemplo. Es interesante, por tanto, estudiar este posicionamiento situando un control para comprender cómo pueden variar los valores en función de la estructura del objeto `Grid` (filas, columnas y posicionamiento del control dentro de ellas).

Padding

La propiedad `Padding` permite definir una zona, situada en el interior de un control, en la que no se mostrará el contenido. Para el control `Button`, por ejemplo, esta propiedad puede utilizarse para que el contenido no se superponga jamás sobre los bordes del control. Su valor se define de la misma manera que el valor de la propiedad `Margin`.

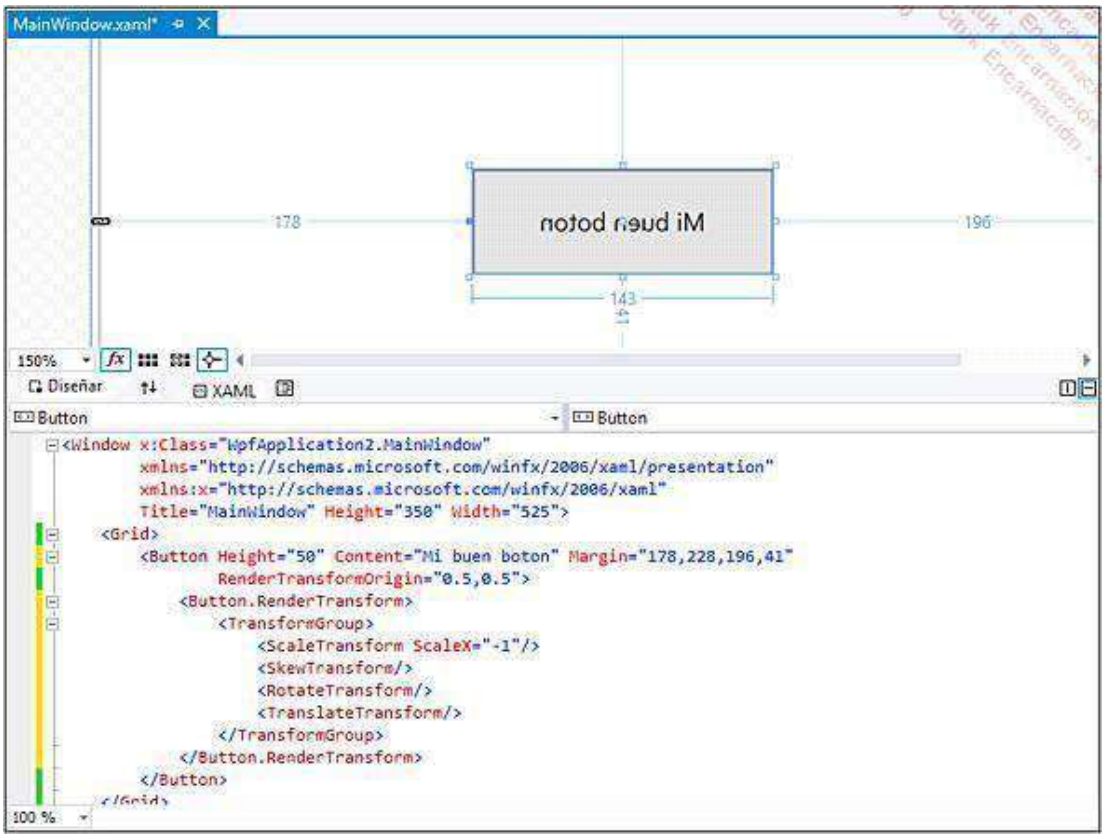
La posición en un Canvas

Cuando un contenedor es de tipo `Canvas`, el posicionamiento se realiza asignando valor a las propiedades `Canvas.Top`, `Canvas.Left`, `Canvas.Right` y `Canvas.Bottom` sobre cada control que contiene. Permiten especificar la distancia entre el borde del `Canvas` y el borde asociado al control. Aceptan valores numéricos, con o sin unidad de medida asociada.

Estas propiedades de posicionamiento son algo particulares, pues pertenecen al control `Canvas`, pero operan sobre los controles que este contiene. Se las denomina **propiedades asociadas**. En particular, puede crearlas para extender las funcionalidades de un control sin tener que heredar de él.

El diseñador visual permite modificar la posición de los controles de manera muy simple. En efecto, arrastrando y soltando un control que ya está situado puede desplazarlo y, de este modo, modificar el valor de su propiedad `Margin`o de las propiedades asociadas propias del control `Canvas`, en función del contenedor utilizado.

También es posible redimensionar un control en modo de diseño. Para ello, basta con seleccionar un control y, a continuación, posicionar el cursor del ratón sobre alguna de las anclas situadas sobre los bordes para estirarla manteniendo pulsado el botón izquierdo del ratón y desplazando el cursor. El diseñador no le impide desplazar el borde derecho más allá del borde izquierdo del control. Considera, en efecto, que este comportamiento es equivalente a darle la vuelta al control y genera una transformación que permite visualizar el control "del revés".



3. Agregar un controlador de eventos a un control

Los controles visuales pueden producir numerosos eventos sobre los que es posible suscribirse desde el código C# asociado a la ventana. Es posible, también, suscribirse a estos eventos desde el código fuente XAML.

Para ello, basta con agregar a la declaración XAML de un control un atributo que tenga el siguiente aspecto:

```
<nombre del evento>="<nombre del controlador>"
```

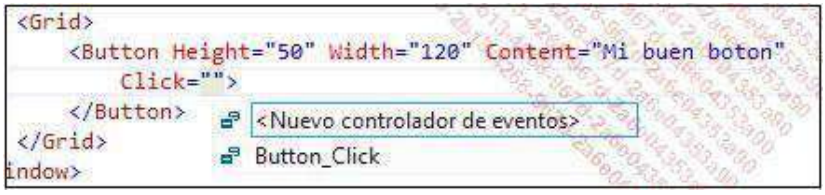
```
<Grid>
    <Button Height="50" Width="120" Content="Mi buen boton"
        Click="Button_Click">
    </Button>
</Grid>
```

El controlador de eventos correspondiente debe encontrarse en el archivo de code-behind de la ventana en edición. Aquí, la gestión del clic sobre el botón se realiza mediante el método `Button_Click` cuya definición es la siguiente.

```
private void Button_Click(object sender, RoutedEventArgs e)
{


}
```

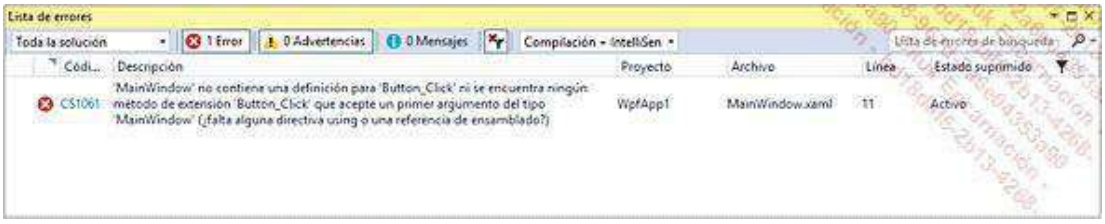
IntelliSense provee una ayuda preciosa para realizar esta operación. Permite, en efecto, generar automáticamente

un controlador correspondiente al tipo de delegado asociado al evento. Permite a su vez, siempre que es posible, asociar el evento a un controlador de eventos existente.



Haciendo doble clic sobre un control se genera también un controlador para el evento por defecto del control. El controlador y el evento se asocian automáticamente en el código XAML.

Cuando un controlador que no existe en el código C# se vincula con un evento en el código XAML, se produce un error de compilación.



Los principales controles

El framework .NET provee, de manera estándar, numerosos controles WPF que cubren la mayoría de necesidades relativas a la creación de una interfaz gráfica moderna. Están definidos en la librería `PresentationFramework.dll`, en el espacio de nombres `System.Windows.Controls`.

La mayoría de estos controles poseen un juego de propiedades comunes que permiten gestionar su representación visual en la aplicación, así como su aspecto. A continuación se enumeran los principales:

`Height` y `Width`	Definen la altura y anchura del control.
`HorizontalAlignment` y `VerticalAlignment`	Definen cómo debe situarse el control en el espacio que tiene disponible horizontal y verticalmente. Los valores pueden ser `Left` (alineación a la izquierda), `Center` (centrado), `Right` (alineación a la derecha) y `Stretch` (ocupar todo el espacio disponible) para la alineación horizontal, y `Top` (alineación arriba), `Bottom` (alineación abajo), `Center` y `Stretch` para la alineación vertical.
`Margin`	Define el espacio que debe quedar libre alrededor de los límites del control.
`Padding`	Define el espacio que debe quedar libre entre el borde del control y su contenido.
`BorderBrush`	Define el color del borde del control.
`BorderThickness`	Define el grosor del borde del control.
`Visibility`	Define el estado de visibilidad del control. Existen tres valores disponibles: `Visible`, `Hidden` (el control se oculta pero la zona que debería ocupar está disponible), `Collapsed` (el control se oculta y la zona que debería ocupar se libera).

Los controles WPF pueden agruparse en varias categorías definidas en función de su objetivo.

1. Controles de ventanas

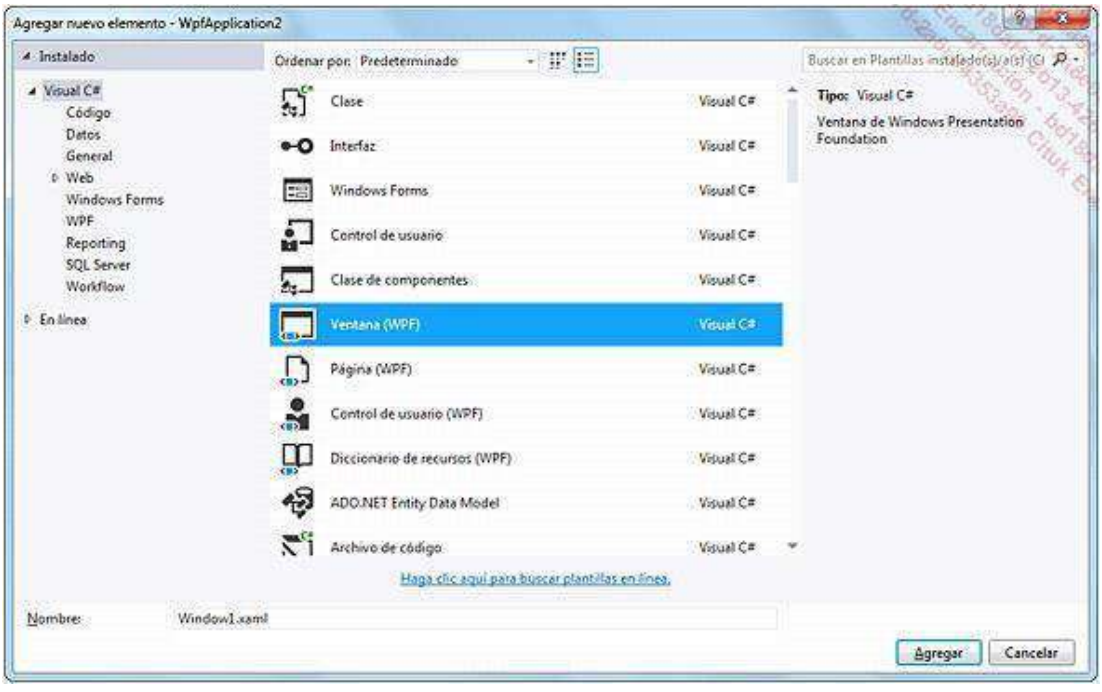
Los controles de ventanas son esenciales en una aplicación WPF. Son ellos los que contienen la interfaz gráfica que va a crear.

Existen dos controles estándar que pertenecen a esta categoría y se corresponden con modos de navegación distintos.

a. Window

El control `Window` permite, como su propio nombre indica, definir una ventana de aplicación. Este control de ventanas es el que se utiliza con mayor frecuencia.

Puede crearse mediante el cuadro de diálogo **Agregar nuevo elemento** (haciendo clic con el botón derecho sobre el proyecto y seleccionando, a continuación, **Agregar - Nuevo elemento** o mediante el atajo de teclado [Ctrl] [Mayús] A):



Al validar la opción **Ventana (WPF)** haciendo clic en el botón **Agregar** se crean un archivo `.xaml` y un archivo `.xaml.cs` que se integran en el proyecto en curso.

El contenido del archivo `.xaml` que acaba de crearse es el siguiente:

```
<Window x:Class="WpfApplication1.Window1"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Window1" Height="300" Width="300">
    <Grid>

    </Grid>
</Window>
```

Como hemos visto antes, el código de este archivo está formado por etiquetas. El objeto `Window` es la raíz del árbol XAML.

El atributo `x:Class` de este objeto se corresponde con la clase C# asociada a la ventana. Esta clase se encarga de lanzar la inicialización de los controles gráficos mediante su constructor. Es también en esta clase donde se asocian los controladores de eventos con los correspondientes controles de la ventana.

El control posee varias propiedades que permiten definir su aspecto, su comportamiento y su ubicación. La siguiente tabla muestra con detalle los principales.

`Title`	Define el título de la ventana.
`ResizeMode`	Define los modos de redimensionamiento disponibles para la ventana.
`ShowInTaskbar`	Define si la ventana debe ser visible en la barra de tareas de Windows.
`WindowState`	Define si la ventana debe poder maximizarse, minimizarse o dimensionarse normalmente.

WindowStartupLocation	Define si la ventana debe estar centrada respecto a su ventana madre o respecto a la pantalla.
Topmost	Define si la ventana debe quedarse en primer plano incluso aunque la aplicación no tenga el foco.

Para abrir una nueva ventana se instancia su método Show, o bien su método ShowDialog si la ventana debe ser modal.

```
Window1 window = new Window1();
window.Show();
```

Este control define también varios eventos, entre ellos Loaded, que permite ejecutar una sección de código cuando el contenido de la ventana se ha cargado, pero todavía no se está mostrando. Los eventos Closing y Closed se producen respectivamente antes y después de cerrar la ventana. El primero permite impedir el cierre de la ventana mediante su controlador de la siguiente manera:

```
private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
{
    e.Cancel = true;
}
```

La ventana de arranque de una aplicación se define en el archivo App.xaml. Una de las propiedades del objeto Application descrita en este archivo es StartupUri. Recibe como valor la ruta relativa que indica qué ventana debe abrirse tras el inicio del programa.

```
<Application x:Class="WpfApplication2.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

b. NavigationWindow

WPF proporciona, además, un modo de multiventana tradicional, un modo de navegación parecido al de un navegador web. En este modo, el usuario no tiene varias ventanas a su disposición, sino una única ventana, dentro de la cual puede navegar entre varias páginas.

La ventana principal es de tipo NavigationWindow y los distintos contenidos se sitúan en objetos Page. Como ocurre en un navegador web, existe un objeto NavigationWindow que dispone de dos botones **Anterior** y **Siguiente** que permite gestionar un histórico de navegación entre cada objeto Page.

NavigationWindow es una clase derivada de la clase Window que tiene acceso a todas las propiedades que hemos visto antes. Posee también propiedades propias de su modo de navegación.

Source	Define la URI de la página que se muestra actualmente.
BackStack	Contiene el histórico de las páginas anteriores.
ForwardStack	Contiene la lista de páginas siguientes.
CanGoBack	Define si es posible, para la ventana, navegar a una página anterior.
CanGoForward	Define si es posible, para la ventana, navegar a una página posterior.

La creación de una NavigationWindow requiere cierta manipulación, pues no existe ninguna plantilla para este tipo de objeto en Visual Studio:

- Abra la ventana **Agregar un nuevo elemento** (haga clic con el botón derecho sobre el proyecto y, a continuación, **Agregar - Nuevo elemento** o [Ctrl][Mayús] A) y cree una ventana (WPF).
- En el archivo .xaml creado, modifique la declaración del objeto raíz: Window se convierte en NavigationWindow. Elimine también las etiquetas <Grid> y </Grid> contenidas en el objeto NavigationWindow. En efecto, este componente no admite el contenido directo.
- En el archivo .xaml.cs asociado modifique el nombre de la clase madre de la misma manera.

```
public partial class Window1 : Window
```

se convierte en:

```
public partial class Window1 : NavigationWindow
```

Llegados a este punto, es posible utilizar la ventana. Es posible, por tanto, asignar un valor a la propiedad StartupUri del objeto Application definido en App.xaml con la ruta relativa que apunta a la ventana que acabamos de crear.

```
StartupUri="Window1.xaml"
```

La creación de un objeto Page puede, en cambio, realizarse mediante una plantilla. En la ventana **Agregar nuevo elemento**, esta plantilla se llama **Página (WPF)**. Agrega un archivo Page1.xaml al proyecto, así como un archivo de code-behind asociado.

El código XAML de esta página es relativamente parecido al de una nueva ventana:

```
<Page x:Class="WpfApplication2.Page1"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
    Title="Page1">
    <Grid>
```

```
    </Grid>
</Page>
```

El contenido de la página se mostrará en la `NavigationWindow` cuando la propiedad `Source` de la ventana tenga como valor "Page1.xaml". Es posible modificar este valor asignando la ruta de un archivo o utilizando el método `Navigate` de la ventana. Este método recibe como parámetro la ruta relativa del archivo `.xaml` que define una página.

```
NavigationWindow ventana = new NavigationWindow();
ventana.Navigate("Page1.xaml");
```

2. Controles de diseño

Los controles de diseño son contenedores que permiten definir el posicionamiento de cada uno de los controles gráficos que contienen. Existen varios tipos de controles de diseño en el framework .NET que responden a la gran mayoría de necesidades en términos de definición de interfaz gráfica.

a. Grid

El control `Grid` representa una **cuadrícula** (o matriz) en la que se definen **filas** y **columnas**. Cada uno de los componentes gráficos se ubica en esta matriz mediante un índice de fila y de columna que puede extenderse sobre varias filas y/o columnas.


El control `Grid` es el contenedor principal por defecto definido en el modelo de ventanas WPF de Visual Studio.

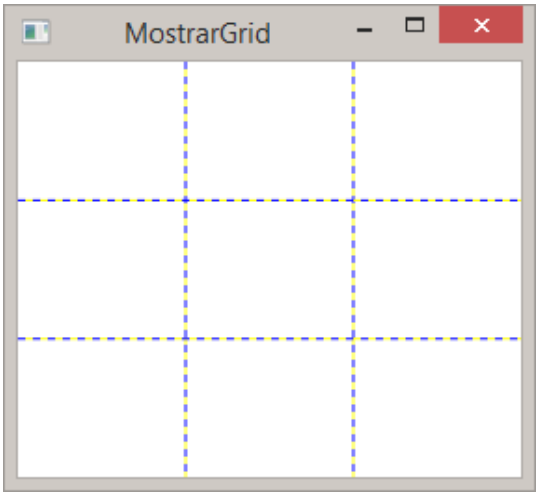
Para definir una cuadrícula con tres filas y tres columnas se escribe el siguiente código:

```
<Grid>
     <Grid.RowDefinitions>
         <RowDefinition />
         <RowDefinition />
         <RowDefinition />
     </Grid.RowDefinitions>
     <Grid.ColumnDefinitions>
         <ColumnDefinition />
         <ColumnDefinition />
         <ColumnDefinition />
     </Grid.ColumnDefinitions>
</Grid>
```

Las propiedades `RowDefinitions` y `ColumnDefinitions` del control `Grid` son colecciones que pueden contener, respectivamente, tantos objetos `RowDefinition` o `ColumnDefinition` como sea necesario.

En tiempo de ejecución, la ventana que contiene esta definición de cuadrícula puede aparecer vacía… En efecto, por defecto nuestra cuadrícula es transparente y está vacía. Para visualizar la cuadrícula es posible definir su propiedad `ShowGridLines` a `True`.

```
<Grid ShowGridLines="True">
    ...
</Grid>
```



Es posible, evidentemente, definir la altura de cada una de las filas mediante la propiedad `Height` de cada objeto `RowDefinition` y el ancho de cada columna mediante la propiedad `Width` de los objetos de tipo `ColumnDefinition`.

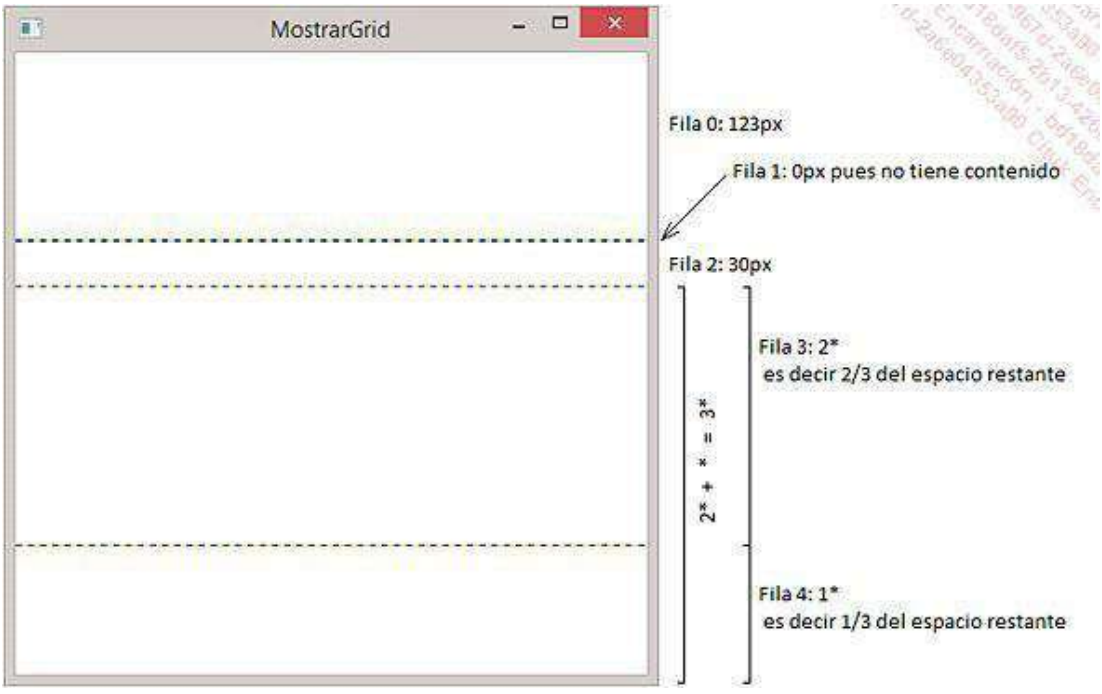
Existen tres formas de especificar estos tamaños, que son, por orden de prioridad:

- Un valor en número de píxeles.
- `Auto`: la fila (o columna) se adapta a su contenido. Si no tiene contenido, no será visible.
- * (Star): el uso de este modo de medida es algo más complejo. Se asocia un coeficiente a cada medida "Star". Este coeficiente vale 1 si no se especifica otra cosa de manera explícita. El tamaño efectivo se calcula según la siguiente fórmula:

$$\text{Tamaño} = \text{Tamaño libre restante} \times \frac{\text{Coeficiente para la fila / columna}}{\text{Suma de los coeficientes de las filas / columnas}}$$

A continuación se muestra un ejemplo que muestra la aplicación de los distintos tipos de medida.

```
<Grid ShowGridLines="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="123"  />
        <RowDefinition  Height="Auto"/>
        <RowDefinition  Height="30"/>
        <RowDefinition  Height="2*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
</Grid>
```



Para posicionar controles en la cuadrícula es necesario declararlos y dotar de valor a cada una de las propiedades asociadas `Grid.Row` y `Grid.Column`. El valor por defecto de estas propiedades es 0, lo cual quiere decir que si no se utilizan de manera explícita en un control, este se posicionará en la primera fila y la primera columna de la cuadrícula.

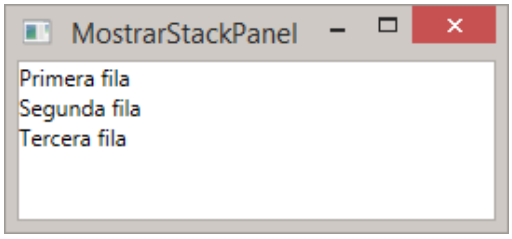
```
<Grid ShowGridLines="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="123"  />
        <RowDefinition  Height="Auto"/>
        <RowDefinition  Height="30"/>
        <RowDefinition  Height="2*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <!-- Insertamos un control Grid en la segunda fila
(index 1) -->
    <Grid Background="Red" Grid.Row="1" Height="50" />
</Grid>
```

b. StackPanel

El control `StackPanel` permite organizar vertical u horizontalmente varios componentes visuales. Su propiedad `Orientation` controla el sentido en el que se apilarán los elementos y su valor por defecto es `Vertical`.

```
<StackPanel Orientation="Vertical">
    <TextBlock Text="Primera fila"  />
    <TextBlock Text="Segunda fila"  />
    <TextBlock Text="Tercera fila"  />
</StackPanel>
```

La representación gráfica de este segmento es la siguiente:

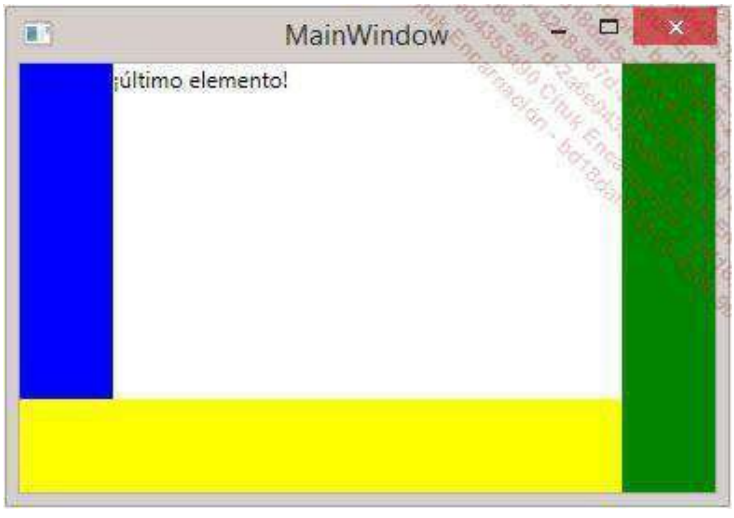


c. DockPanel

El control `DockPanel` ofrece la posibilidad de anclar sus controles hijos sobre cada uno de sus cuatro lados. Para ello utiliza la propiedad asociada `DockPanel.Dock` sobre los controles que se desea anclar.

```
<DockPanel>
    <Grid DockPanel.Dock="Right" Background="Green" Width="50" />
    <Grid DockPanel.Dock="Bottom" Background="Yellow" Height="50" />
    <Grid DockPanel.Dock="Left" Background="Blue" Width="50" />
    <TextBlock Text="¡último elemento!" />
</DockPanel>
```

Este código produce el siguiente resultado:

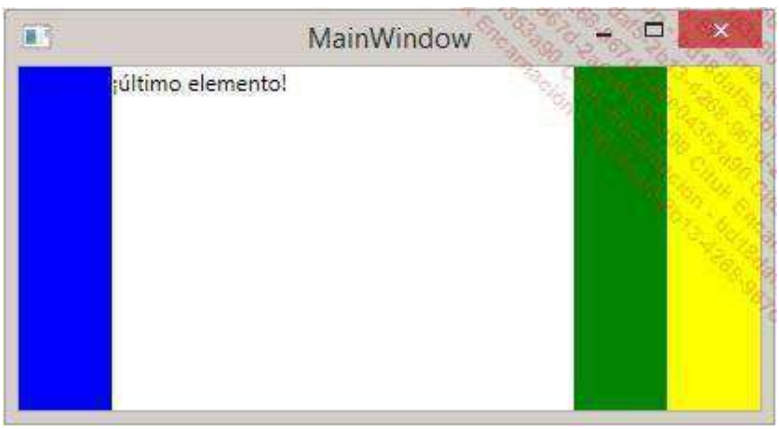


Es importante destacar que los controles se anclan en función de su orden de instanciación. Esto implica que el código anterior y el siguiente ejemplo no produzcan el mismo resultado.

```
<DockPanel>
    <Grid DockPanel.Dock="Bottom" Background="Yellow" Height="50" />
    <Grid DockPanel.Dock="Right" Background="Green" Width="50" />
    <Grid DockPanel.Dock="Left" Background="Blue" Width="50" />
    <TextBlock Text="¡último elemento!" />
</DockPanel>
```

Es posible anclar varios controles al mismo borde: éstos se apilarán de manera relativa respecto a este borde, en orden de instanciación.

```
<DockPanel>
    <Grid DockPanel.Dock="Right" Background="Yellow" Width="50" />
    <Grid DockPanel.Dock="Right" Background="Green" Width="50" />
    <Grid DockPanel.Dock="Left" Background="Blue" Width="50" />
    <TextBlock Text="¡último elemento!" />
</DockPanel>
```

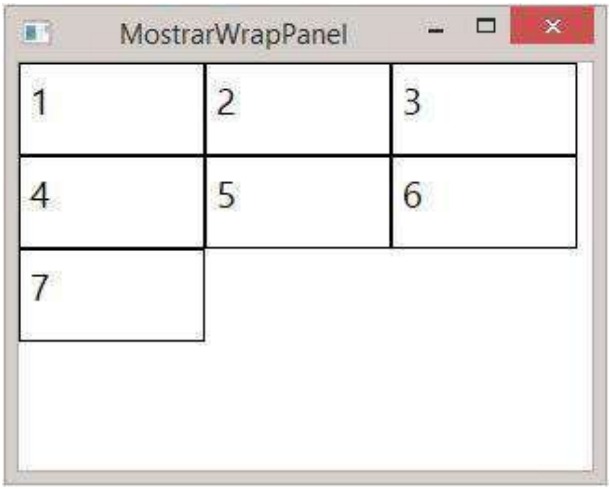


La clase `DockPanel` tiene la propiedad `LastChildFill` que permite configurar cuál debe ser el comportamiento de su último control hijo. Si esta propiedad tiene el valor `True` entonces el elemento cubrirá la totalidad del espacio restante en el `DockPanel`.

d. WrapPanel

El control `WrapPanel` podría describirse como un `StackPanel` mejorado (si bien la clase `WrapPanel` no hereda de `StackPanel`). En efecto, del mismo modo que el control `StackPanel`, permite organizar varios componentes gráficos.

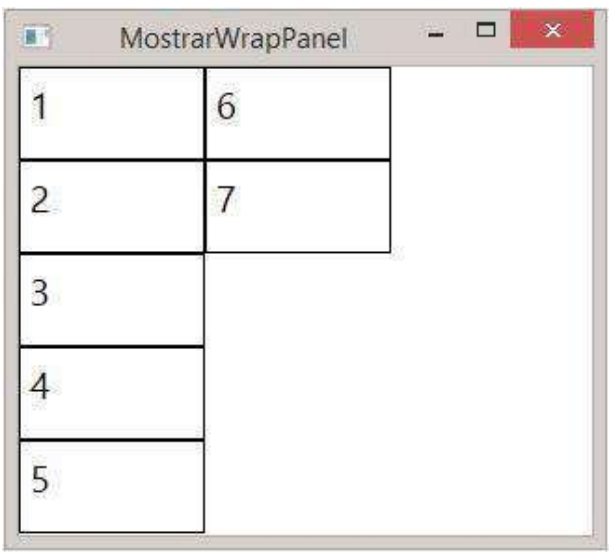
La verdadera diferencia entre ambos controles reside en el hecho de que el `WrapPanel` organiza el contenido en una nueva fila (o columna) cuando el espacio disponible es insuficiente para la representación deseada.



Esta pantalla es el resultado del siguiente código XAML:

```
<WrapPanel>
    <Label Content="1" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="2" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="3" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="4" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="5" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="6" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="7" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
</WrapPanel>
```

Por defecto, la propiedad `Orientation` del `WrapPanel` es `Horizontal`. Cambiando este valor a `Vertical` se obtiene el siguiente resultado:

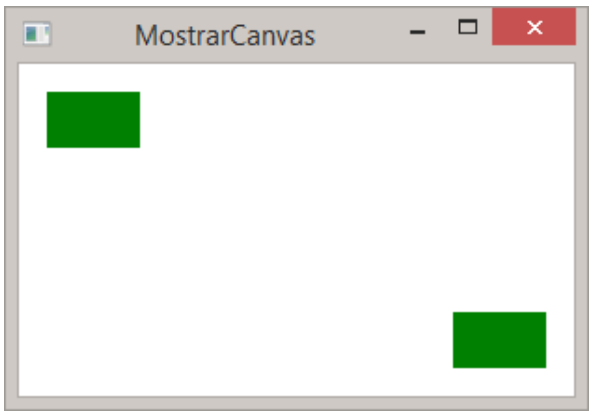


El control `WrapPanel` ocupa, por defecto, todo el espacio disponible en su control padre, lo que permite tener un comportamiento dinámico: si se redimensiona el elemento padre en tiempo de ejecución, el `WrapPanel` se redimensiona también automáticamente y sus elementos hijo pueden reposicionarse en función de las nuevas dimensiones.

e. Canvas

El control `Canvas` está pensado para contener controles gráficos posicionados de manera absoluta.

El posicionamiento de los elementos se implementa mediante las propiedades asociadas correspondientes `Canvas.Top`, `Canvas.Left`, `Canvas.Right` y `Canvas.Bottom`. Estas propiedades permiten, respectivamente, especificar la distancia entre el control posicionado y el borde del control `Canvas`. El siguiente código posiciona dos rectángulos, uno a 15 píxeles de los bordes izquierdo y superior, y el otro a 15 píxeles de los bordes derecho e inferior.




```
<Canvas>
    <Rectangle Width="50" Height="30" Fill="Green"
Canvas.Top="15" Canvas.Left="15" />
    <Rectangle Width="50" Height="30" Fill="Green"
Canvas.Right="15" Canvas.Bottom="15" />
</Canvas>
```

3. Controles de representación de datos

Cualquier aplicación gráfica necesita mostrar datos e información, bien sean de tipo texto o imagen, temporales o permanentes. En WPF existen varios controles disponibles para gestionar estos casos.

a. TextBlock

El control `TextBlock` es el control mejor adaptado para representar texto. Su contenido se especifica mediante la propiedad `Text`.

```
<TextBlock Text="Desarrollo con VS2017 y C#" />
```

Existen otras propiedades que permiten controlar la manera en la que se representa el texto: color, tipo de letra o

incluso la gestión de los saltos de línea.

Las principales se enumeran en la siguiente tabla:

`Text`	Define el texto que se muestra en el control.
`FontFamily`	Define el tipo de letra utilizado en el control.
`FontSize`	Define el tamaño de la letra para mostrar el contenido.
`FontStyle`	Define el estilo de la letra: cursiva, normal u oblicua (cursiva simulada transformando el tipo de letra si no existe una versión en cursiva).
`FontWeight`	Define el grosor de la fuente (espesor del texto) utilizada para el contenido.
`Foreground`	Define el color del texto.
`TextAlignment`	Define la alineación del texto (izquierda, derecha, centrada, justificada).

```
<TextBlock Text="Desarrollo con VS2017 y C#" FontSize="30"
FontFamily="Segoe Print"/>
```

b. Label

El control `Label`, en su uso más simple, es similar a un control `TextBlock`. No tiene propiedad `Text` sino una propiedad `Content`. El motivo de esta diferencia es el hecho de que el control `Label` puede contener cualquier tipo de control.

Este control implementa también una funcionalidad bastante interesante: permite asignar el foco a un control asociado mediante las Access Keys.

Las Access Keys son combinaciones de teclas que permiten acceder a un control. En Windows, esta combinación implica, generalmente, el uso de la tecla [Alt] y algún carácter alfabético. Para utilizar esta funcionalidad es preciso:

- Definir un `Label` y asignar valor a su propiedad `Content` con una cadena de caracteres cuya letra esté precedida por el símbolo _. Esta letra será, a continuación, la que se utilizará en la combinación de teclas [Alt] + [letra].
- Definir un objeto que se asociará al `Label`. Este objeto recibirá el foco cuando se utilice la Access Key.
- Vincular el `Label` y el objeto asociado mediante la propiedad `Target` del `Label` utilizando el binding siguiente:

```
Target="{Binding ElementName=<nombre del objeto asociado>}"
```

```
<Grid>
  <Grid.RowDefinitions>
      <RowDefinition Height="*" />
```

```
      <RowDefinition Height="*" />
  </Grid.RowDefinitions>

  <StackPanel Grid.Row="0" Orientation="Horizontal" Height="25">
   <Label Content="_Apellido" Target="{Binding ElementName=tbApellido}" />
   <TextBox x:Name="tbApellido" Width="150" />
  </StackPanel>

  <StackPanel Grid.Row="1" Orientation="Horizontal" Height="25">
     <Label Content="_Nombre" Target="{Binding ElementName=tbNombre}" />
     <TextBox x:Name="tbNombre" Width="150" />
  </StackPanel>
</Grid>
```

Tras la ejecución, si pulsamos sobre la tecla [Alt] vemos que se resaltan las Access Keys que podemos utilizar.

Aquí, la combinación de teclas [Alt] A pone el foco sobre el campo de edición Apellido, mientras que si pulsamos simultáneamente sobre [Alt] y N se asigna el foco al campo de edición Nombre.

c. Image

El control `Image` permite visualizar, entre otros, el contenido de archivos BMP, JPEG o PNG.

Su propiedad `Source` permite definir la ubicación de la imagen que se desea mostrar. Esta ubicación puede ser una ruta absoluta o relativa, una URI que apunte sobre alguno de los recursos de la aplicación o incluso una URL. En este último caso, la descarga de la imagen se realiza automáticamente antes de mostrarla.

La propiedad `Stretch` de este control define la manera en que se estirará la imagen en función del espacio disponible.

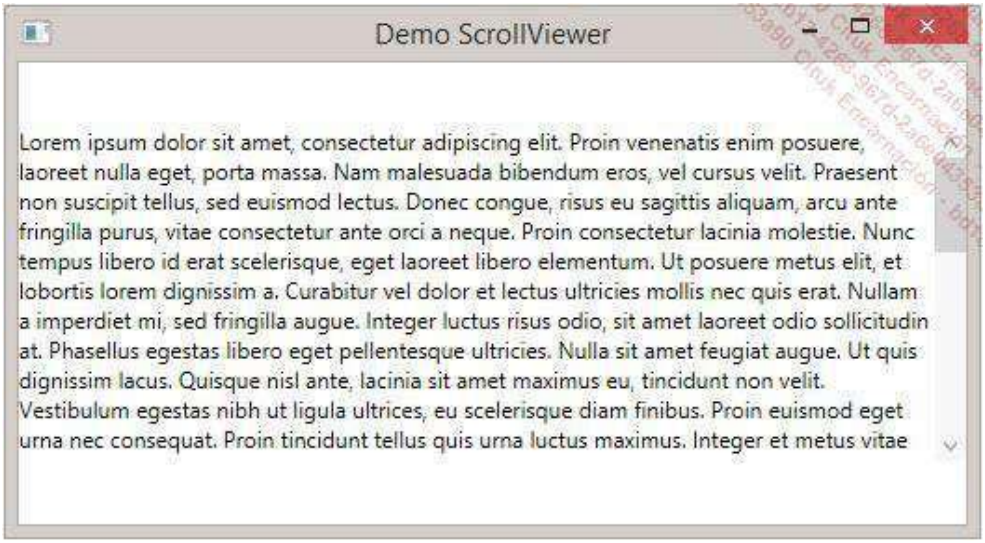
```
<Image Source="http://upload.wikimedia.org/wikipedia/commons/a/af/
Jimi_Hendrix_1967_uncropped.jpg" Stretch="Uniform" />
```

Este código produce el siguiente resultado:

d. ScrollViewer

Algunos controles necesitan más espacio que el que se les ha asignado. Habitualmente, este tipo de casos se gestiona mediante el uso de barras de desplazamiento horizontales o verticales, pero ciertos controles WPF no disponen, de manera nativa, de estas barras. El control `ScrollViewer` es una implementación externa de estas barras que permite situarlas en cualquier control para habilitar el desplazamiento.

```
<ScrollViewer Height="120" >
    <TextBlock TextWrapping="Wrap" Text="Escriba aquí un texto muy
muy largo..." />
</ScrollViewer>
```



e. ItemsControl

Es posible visualizar una lista de elementos mediante un objeto de tipo `ItemsControl`. Este objeto puede contener una colección de objetos de cualquier tipo. Estos objetos se representan por defecto como una lista vertical, sin barra de desplazamiento. Cuando la representación de la lista es más grande que el control, la lista se trunca. Para gestionar este caso de uso, conviene situar el control de la lista dentro de un objeto de tipo `ScrollViewer`.

Los valores a representar se proveen al control mediante su propiedad `Items`.

```
<ItemsControl Height="30">
    <ItemsControl.Items>
        <system:String>Opción  n°1</system:String>
        <system:String>Opción  n°2</system:String>
        <system:String>Opción  n°3</system:String>
        <system:String>Opción  n°4</system:String>
    </ItemsControl.Items>
</ItemsControl>
```

Para agregar objetos de tipo `string` al control, se agrega el espacio de nombres XML `system` a la ventana mediante la siguiente declaración:

```
xmlns:system="clr-namespace:System;assembly=mscorlib"
```

La propiedad `Items`, al ser la propiedad por defecto del control, permite condensar el código omitiendo las declaraciones de apertura y cierre correspondientes a esta propiedad:

```
<ItemsControl Height="30">
    <system:String>Opción  n°1</system:String>
    <system:String>Opción  n°2</system:String>
    <system:String>Opción  n°3</system:String>
    <system:String>Opción  n°4</system:String>
</ItemsControl >
```

Es frecuente que los elementos contenidos por el control se definan de manera dinámica. Para ello, es preferible utilizar la propiedad `ItemsSource` asignándole valor con la declaración de un binding.

```
<ItemsControl Height="30" ItemsSource="{Binding ListaDeOpciones}" />
```

En ambos casos, cada elemento de la colección `ListaDeOpciones` genera un elemento en la lista visual. Ambas están vinculadas: el contexto de datos del elemento visual es el elemento correspondiente de la lista `ListaDeOpciones`.

Esta relación es importante cuando los datos proporcionados son de tipo complejo. La representación visual por defecto de cada elemento está determinada por el uso de su método `ToString`, aunque el resultado visual no siempre es el deseado.

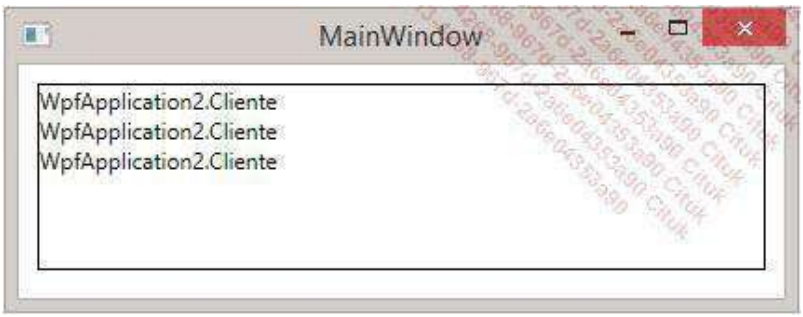
El siguiente ejemplo está basado sobre el tipo de datos definido a continuación.

```
public class Cliente
{
    public string Apellidos { get; set; }
    public string Nombre { get; set; }
    public decimal ImporteTotalCompras { get; set; }
}
```

```
<ItemsControl VerticalAlignment="Top" BorderBrush="Black"
BorderThickness="1" Margin="10" Height="150">
    <local:Cliente Apellidos="MARTIN FUENTES" Nombre="Juan"
ImporteTotalCompras="127.42" />
    <local:Cliente Apellidos="HERNANDEZ MARTIN" Nombre="Eric"
ImporteTotalCompras="98.02" />
    <local:Cliente Apellidos="ALCALDE BLANCO" Nombre="Sofía"
ImporteTotalCompras="241.95" />
</ItemsControl>
```

El espacio de nombres `local` se define previamente en la raíz del archivo.

La representación visual del control es la siguiente:



Para mejorar la representación visual de los datos se utiliza la propiedad `ItemTemplate` del control. Esta propiedad acepta como valor un objeto del tipo `DataTemplate`, que define un modelo visual para el contexto de datos.

El siguiente código XAML define una cuadrícula que contiene dos filas y dos columnas en las que se sitúan las tres propiedades de la clase `Cliente` mediante expresiones de binding.

```
<ItemsControl VerticalAlignment="Top" BorderBrush="Black"
BorderThickness="1" Margin="10" Height="150">
    <local:Cliente Apellidos="MARTIN FUENTES" Nombre="Juan"
ImporteTotalCompras="127.42" />
    <local:Cliente Apellidos="HERNANDEZ MARTIN" Nombre="Eric"
ImporteTotalCompras="98.02" />
    <local:Cliente Apellidos="ALCALDE BLANCO" Nombre="Sofía"
ImporteTotalCompras="241.95" />

    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition />
                    <ColumnDefinition />
                </Grid.ColumnDefinitions>
                <Grid.RowDefinitions>
                    <RowDefinition />
```

```
                    <RowDefinition />
                </Grid.RowDefinitions>

                <TextBlock Grid.Row="0" Grid.Column="0"
FontSize="18" Text="{Binding Apellidos}" />
                <TextBlock Grid.Row="1" Grid.Column="0"
FontSize="14" Text="{Binding Nombre}" />
                <TextBlock Grid.Row="0" Grid.Column="1"
Grid.RowSpan="2" FontSize="20" Text="{Binding
ImporteTotalCompras, StringFormat='{}{0:N2} €'}" />
            </Grid>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

El aspecto visual de la interfaz tiene en cuenta el `DataTemplate` definido y adapta la representación en consecuencia, lo cual produce el siguiente resultado:

f. StatusBar

El control `StatusBar` representa una barra de estado que se sitúa, habitualmente, en la parte inferior de una ventana de aplicación. Esta barra contiene información relativa al trabajo en curso: número de página, estado de la copia de seguridad del archivo, etc.

Este control puede contener objetos de tipo `StatusBarItem`, cada uno de los cuales contiene a su vez los controles necesarios para la visualización de la información. Es posible definir una separación entre estos objetos mediante el control `Separator`.

```
<DockPanel LastChildFill="False">
    <StatusBar DockPanel.Dock="Bottom" Height="30">
        <StatusBarItem>Ningún archivo abierto</StatusBarItem>
        <Separator />
        <StatusBarItem>Carga del diseñador en
curso...</StatusBarItem>
    </StatusBar>
</DockPanel>
```

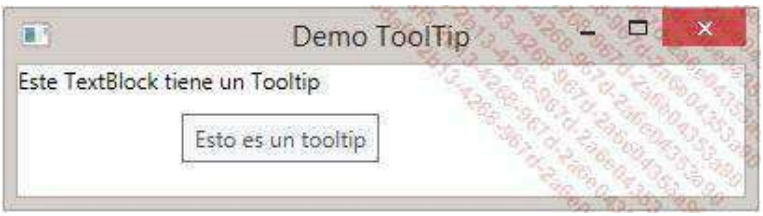
g. ToolTip

El control `ToolTip` (información sobre herramientas) permite mostrar un mensaje de información cuando se pasa el cursor del ratón por encima de un control. Un caso de uso habitual es la visualización de un mensaje explicativo cuando se está introduciendo un dato erróneo (un formato incorrecto de correo electrónico, por ejemplo).

Los componentes gráficos poseen una propiedad `ToolTip` que se corresponde con el contenido que se desea mostrar. Este contenido, en su forma más simple, es una cadena de caracteres, pero también puede ser más complejo y mostrar una imagen con texto complementario, por ejemplo.

```
<TextBlock Text="Este TextBlock tiene un Tooltip">
    <TextBlock.ToolTip>
        <ToolTip>
            <TextBlock Text="Esto es un tooltip" />
        </ToolTip>
    </TextBlock.ToolTip>
</TextBlock>
```

Esta sección de código produce el siguiente resultado:



4. Controles de edición de texto

Los controles de edición de texto son puntos de entrada privilegiados para registrar información por parte de los usuarios. Ofrecen, en efecto, cierta libertad en cuanto al contenido y constituyen verdaderos puntos de creación de datos.

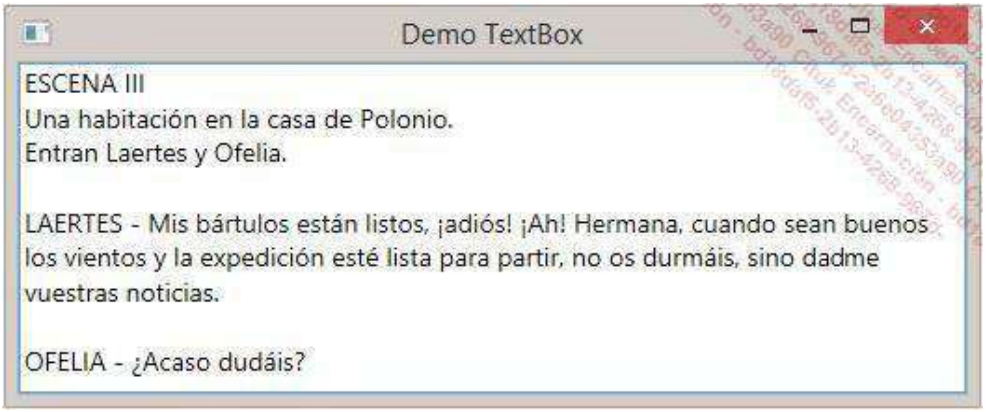
a. TextBox

El control `TextBox` es el control más sencillo y el más utilizado en la edición de texto. Su contenido se define mediante su propiedad `Text`. Por defecto, es imposible saltar de línea en este control.

La propiedad `AcceptsReturn`, cuando se define a `true`, permite aceptar un retorno de línea mediante la tecla [Enter]. Existe otra propiedad que permite insertar un salto de línea automático cuando el texto es más largo que el control: `TextWrapping`. Basta con definir su valor a `Wrap` para obtener el resultado esperado.

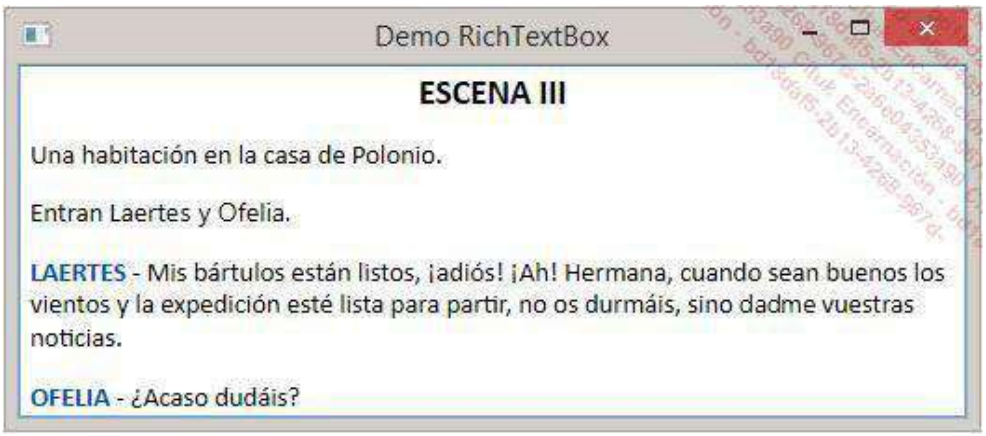
El siguiente código permite obtener el siguiente resultado:

```
<TextBox AcceptsReturn="True" TextWrapping="Wrap" />
```



b. RichTextBox

El texto que se ha presentado en la sección anterior puede resultar difícil de leer, pues todos los elementos de texto están formateados de la misma manera. Para resolver este problema, el control `RichTextBox` permite mostrar texto en formato RTF (como la aplicación WordPad de Windows). Editando este texto en WordPad y realizando un copiar/pegar hacia un componente `RichTextBox`es posible obtener el siguiente resultado:



En este estado, no es posible modificar el formato del texto. En efecto, WPF no provee la barra de herramientas necesaria para un uso avanzado del control `RichTextBox`. No obstante, el control provee las propiedades y métodos necesarios para su extensión.

La propiedad `Selection` permite recuperar la zona de texto seleccionado mediante un objeto de tipo `TextSelection`. Este tipo implementa en particular el método `ApplyPropertyValue (DependencyProperty formattingProperty, object value)` que permite modificar una propiedad del texto seleccionado.

A continuación, agregamos un botón que permite modificar el grosor del texto seleccionado. El archivo `MainWindow.xaml.cs` define un controlador para el evento `Click` sobre el botón definido antes del control `RichTextBox`.

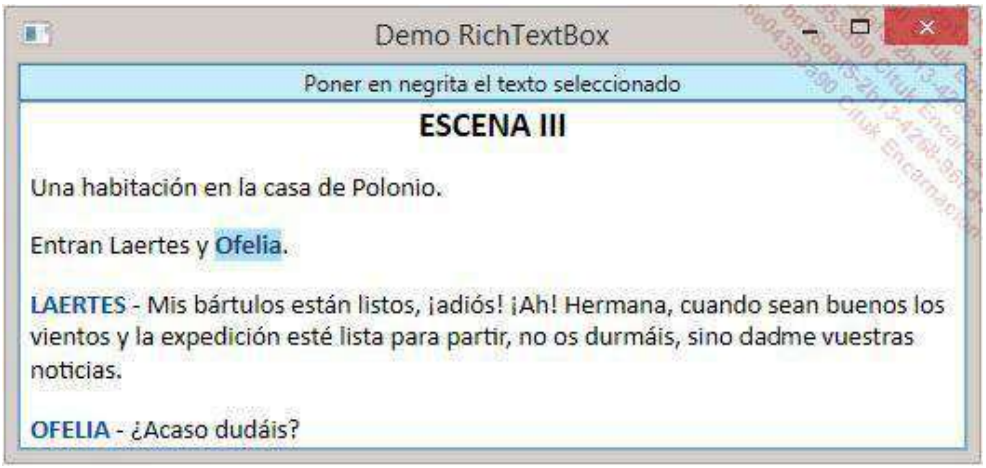
- MainWindow.xaml :

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition />
    </Grid.RowDefinitions>

    <Button Grid.Row="0" Content="Poner en negrita el texto
seleccionado" Click="Button_Click" />
    <RichTextBox Grid.Row="1" AcceptsReturn="True"
x:Name="zonaTexto" />
</Grid>
```

- MainWindow.xaml.cs :

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    zonaTexto.Selection.ApplyPropertyValue(FontWeightProperty,
FontWeights.Bold);
}
```

El resultado es el siguiente:



El funcionamiento del control `Button` se detalla más adelante en este capítulo.

c. PasswordBox

Las reglas de seguridad informática incluyen, todas ellas, la no compartición de contraseñas para evitar cualquier uso no deseado de un componente o aplicación. La primera manera de evitar que una contraseña caiga en las manos indebidas es su ocultación. Esta solución se utiliza, por otro lado, en la práctica totalidad de editores.

Con WPF, la implementación de esta ocultación puede realizarse mediante el control `PasswordBox`. Este control remplaza automáticamente los caracteres escritos por el carácter especificado en su propiedad `PasswordChar`, preservando la privacidad del contenido escrito en su propiedad `Password`.

5. Controles de selección

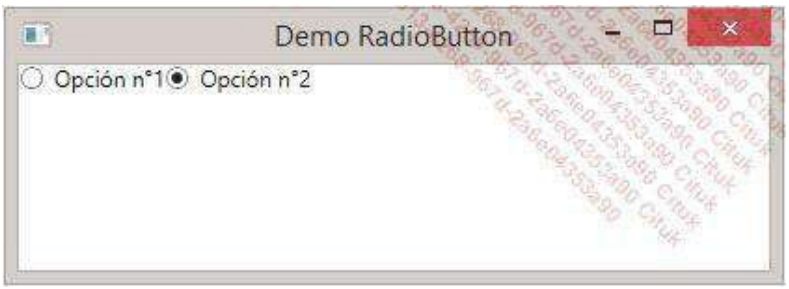
Los controles de selección son mucho menos permisivos que los controles de edición, pues permiten seleccionar entre una o varias opciones de entre una lista finita de elementos. No obstante, este comportamiento puede tener un gran valor, por otro lado, cuando se pretende acelerar la información de un formulario reduciendo las posibilidades de error en las operaciones de categorización o de edición repetitiva.

a. RadioButton

El control `RadioButton` representa una opción de selección exclusiva que puede marcarse o no. Es posible recuperar este estado mediante la propiedad `IsChecked`. Cuando el control está marcado, es imposible desmarcarlo haciendo clic de nuevo en él. Para ello, es preciso marcar algún otro control `RadioButton` que pertenezca al mismo grupo. El grupo al que pertenece el control se define mediante la propiedad `GroupName`.

```
<StackPanel Orientation="Horizontal">
    <RadioButton GroupName="Opción" Content="Opción n°1"  />
    <RadioButton GroupName="Opción" Content="Opción n°2"  />
</StackPanel>
```

Este código produce el siguiente resultado.



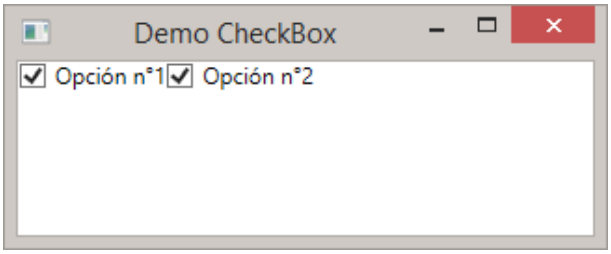
Este control se utiliza con poca frecuencia cuando se muestran más de cinco opciones. En este caso, es más adecuado utilizar el control `ComboBox`, mucho más práctico que una pila de `RadioButton`, y mucho más legible.

b. CheckBox

El control `CheckBox` permite, como el `RadioButton`, seleccionar o no un valor, pero su modo de selección no es exclusivo: es posible marcar varios `CheckBox` al mismo tiempo. El estado de cada `CheckBox` se define mediante el valor de su propiedad `IsChecked`.

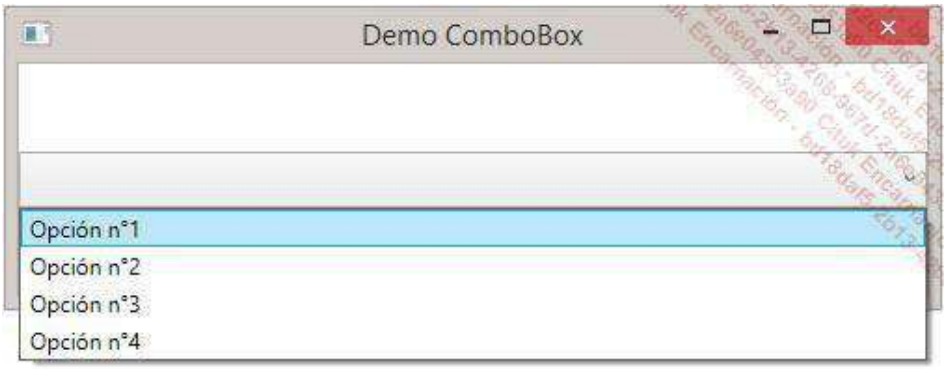
El siguiente código permite mostrar dos controles `CheckBox` :

```
<StackPanel  Orientation="Horizontal">
    <CheckBox Content="Opción n°1"  />
    <CheckBox Content="Opción n°2"  />
</StackPanel>
```



c. ComboBox

Este control proporciona una lista desplegable de elementos donde solamente es posible seleccionar uno. Este control resulta práctico para realizar selecciones únicas debido a lo sencillo que resulta su uso y su tamaño, mínimo.



Este control deriva del tipo `ItemsControl`, lo cual implica que su uso sea algo similar. Los datos que se muestran se proveen mediante propiedades `Items` o `ItemsSource`. Esta segunda propiedad se alimenta mediante una expresión de tipo binding.

Del mismo modo que con el tipo `ItemsControl`, es posible personalizar la presentación de los elementos que se muestran asignando valor a la propiedad `ItemTemplate` del control mediante un objeto de tipo `DataTemplate`.

El uso de `DataTemplate` para la personalización de la representación gráfica se detalla al mismo tiempo que el tipo `ItemsControl` (consulte la sección ItemsControl).

Cuando se selecciona un elemento, su valor se registra en la propiedad `SelectedItem`del control.

d. ListBox

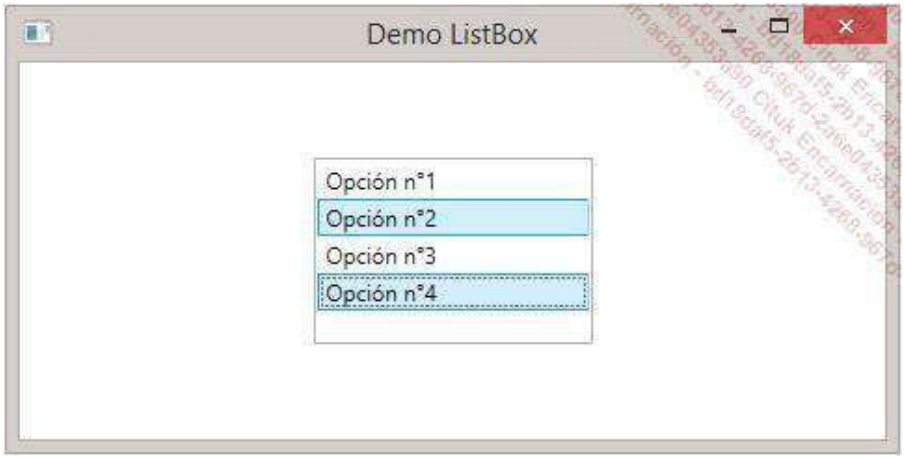
El control `ListBox` muestra una lista de elementos en la que es posible seleccionar uno o varios. El modo de selección se define mediante el valor de la propiedad `SelectionMode` del control: `Single`, que permite seleccionar un único valor, o bien `Multiple` y `Extended`, que permiten realizar una selección múltiple. La diferencia entre estos dos últimos modos es la siguiente:

- `Multiple`: la selección o deselección de un elemento se realiza mediante un clic.
- `Extended`: es necesario mantener pulsada la tecla [Ctrl] durante las operaciones de selección y deselección.

La lista de datos contenidos en el control se provee mediante sus propiedades `Items` o `ItemsSource`, de la misma manera que con el tipo `ItemsControl`. Como con los demás controles que permiten mostrar colecciones,

es posible personalizar la representación gráfica mediante la propiedad `ItemTemplate` con un objeto de tipo `DataTemplate`.

El uso de `DataTemplate` para la personalización de la representación gráfica se detalla al mismo tiempo que el tipo `ItemsControl` (consulte la sección ItemsControl).



Este resultado se obtiene utilizando la siguiente sección de código.

```
<ListBox SelectionMode="Multiple" Height="100" Width="150">
    <ListBox.Items>
        <system:String>Opción  n°1</system:String>
        <system:String>Opción  n°2</system:String>
        <system:String>Opción  n°3</system:String>
        <system:String>Opción  n°4</system:String>
    </ListBox.Items>
</ListBox>
```

Cuando se selecciona un elemento, su valor se almacena en la propiedad `SelectedItem` del control. Si se seleccionan varios valores, es posible recuperarlos mediante la propiedad `SelectedItems`.

e. ListView

El control `ListView`, que deriva del tipo `ItemsControl`, permite mostrar una lista de elementos en una cuadrícula. Esta se define mediante la propiedad `View`, de tipo `ViewBase`. La única clase del framework .NET que hereda de este tipo es la clase `GridView`, por lo que será un objeto de este tipo el que pasaremos a la propiedad `View`.

Es posible, en efecto, definir otro formato de presentación desarrollando un control que herede de `ViewBase`.

Los elementos que se muestran en el control se pasan, como ocurre con el tipo `ItemsControl`, mediante las propiedades `Items` o `ItemsSource`.

El tipo de datos utilizado en este ejemplo se define de la siguiente manera:

```
public class Cliente
{
```

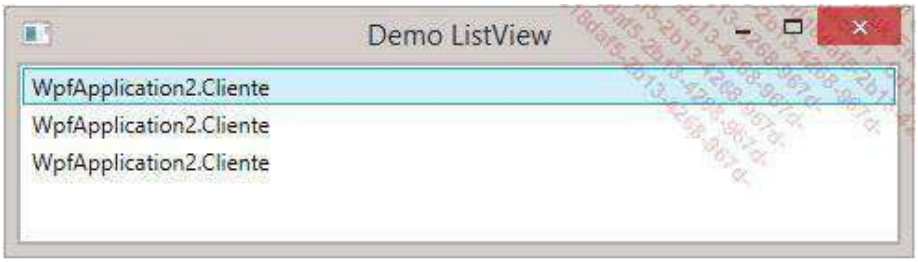
```
    public string Apellidos { get; set; }
    public string Nombre { get; set; }
    public decimal ImporteTotalCompras { get; set;  }
}
```

Una vez declarado este tipo, se agregan elementos a la propiedad `Items` del control `ListView`.

```
<ListView>
    <ListView.Items>
        <local:Cliente Apellidos="MARTIN FUENTES" Nombre="Juan"
ImporteTotalCompras="127.42" />
        <local:Cliente Apellidos="HERNANDEZ MARTIN" Nombre="Eric"
ImporteTotalCompras="98.02" />
        <local:Cliente Apellidos="ALCALDE BLANCO" Nombre="Sofía"
ImporteTotalCompras="241.95" />
    </ListView.Items>
</ListView>
```

El espacio de nombres `local` se define en la declaración del control `Window` padre. El uso de los espacios de nombres se detalla en la primera sección de este capítulo (sección XAML).

Por defecto, el control tiene el mismo aspecto que un control `ListBox`: muestra cada uno de sus elementos ejecutando su método `ToString` como una lista vertical.



La representación gráfica del formato de la cuadrícula se realiza asignando valor a la propiedad `View` mediante un objeto de tipo `GridView`. Esta cuadrícula debe contener una o varias columnas. Para cada columna, se define el valor que se quiere mostrar asignando valor a su propiedad `DisplayMemberBinding` con una expresión de binding.

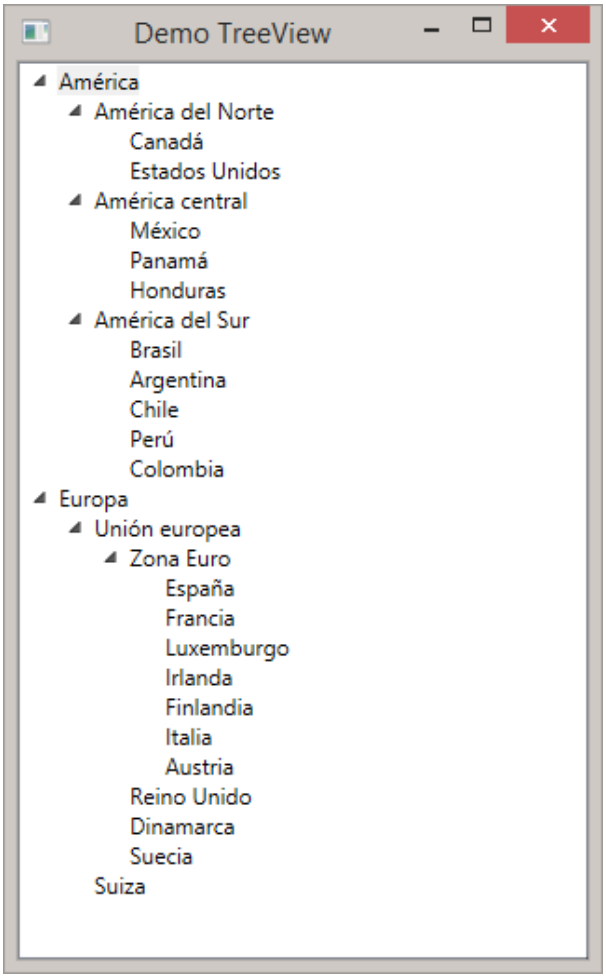
```
<ListView>
    <ListView.Items>
        <local:Cliente Apellidos="MARTIN FUENTES" Nombre="Juan"
ImporteTotalCompras="127.42" />
        <local:Cliente Apellidos="HERNANDEZ MARTIN" Nombre="Eric"
ImporteTotalCompras="98.02" />
        <local:Cliente Apellidos="ALCALDE BLANCO" Nombre="Sofía"
ImporteTotalCompras="241.95" />
    </ListView.Items>
    <ListView.View>
        <GridView>
            <GridViewColumn Header="Apellidos" Width="120"
```

```
DisplayMemberBinding="{Binding Apellidos}" />
            <GridViewColumn Header="Nombre" Width="120"
DisplayMemberBinding="{Binding Nombre}" />
            <GridViewColumn Header="Importe compras" Width="120"
DisplayMemberBinding="{Binding ImporteTotalCompras,
StringFormat={}{0} €}" />
        </GridView>
    </ListView.View>
</ListView>
```

A continuación se muestra el resultado obtenido:

f. TreeView

El control `TreeView` ofrece la posibilidad de mostrar datos de manera jerárquica. Cada uno de los objetos puede tener, en efecto, uno o varios hijos que pueden, ellos mismos, ser padres de otros nodos, y así sucesivamente.



WPF permite describir esta jerarquía de manera sencilla y clara. Basta, en efecto, con considerar que cada nodo de nuestro árbol es un objeto de tipo `TreeViewItem` que puede contener uno o varios objetos del mismo tipo.

El texto correspondiente a cada uno de los nodos se define mediante la propiedad `Header` de cada objeto `TreeViewItem`. Esta propiedad es de tipo `object`, lo que significa, indirectamente, que no es obligatorio darle valor mediante una cadena de caracteres, sino que puede contener también imágenes, texto, una opción a marcar, etc.

El `TreeView` que se mostraba más arriba se codifica de la siguiente manera:

```
<TreeView>
    <TreeViewItem Header="América">
        <TreeViewItem Header="América del Norte">
            <TreeViewItem Header="Canadá" />
            <TreeViewItem Header="Estados Unidos" />
        </TreeViewItem>
        <TreeViewItem Header="América central">
            <TreeViewItem Header="México"  />
            <TreeViewItem Header="Panamá"  />
            <TreeViewItem Header="Honduras" />
        </TreeViewItem>
        <TreeViewItem Header="América del Sur">
            <TreeViewItem Header="Brasil" />
            <TreeViewItem Header="Argentina" />
            <TreeViewItem Header="Chile" />
```

```
            <TreeViewItem Header="Perú" />
            <TreeViewItem Header="Colombia" />
        </TreeViewItem>
    </TreeViewItem>
    <TreeViewItem Header="Europa">
        <TreeViewItem Header="Unión  europea">
            <TreeViewItem Header="Zona  Euro">
                <TreeViewItem Header="España" />
                <TreeViewItem Header="Francia" />
                <TreeViewItem Header="Luxemburgo" />
                <TreeViewItem Header="Irlanda" />
                <TreeViewItem Header="Finlandia" />
                <TreeViewItem Header="Italia" />
                <TreeViewItem Header="Austria" />
            </TreeViewItem>
            <TreeViewItem Header="Reino Unido" />
            <TreeViewItem Header="Dinamarca" />
            <TreeViewItem Header="Suecia" />
        </TreeViewItem>
        <TreeViewItem Header="Suiza" />
    </TreeViewItem>
</TreeView>
```

La clase `TreeViewItem` deriva de la clase `HeaderedItemsControl`, que representa una lista que posee un encabezado. Este encabezado es el elemento que se visualiza como nodo, mientras que la parte que representa a los elementos de la colección está, de hecho, representada visualmente por los nodos hijo. La escritura completa de la declaración de un `TreeViewItem` y de sus hijos es, de hecho, la siguiente:

```
<TreeViewItem Header="América del Norte">
    <TreeViewItem.Items>
        <TreeViewItem Header="Canadá" />
        <TreeViewItem Header="Estados Unidos" />
    </<TreeViewItem.Items>
</TreeViewItem>
```

El uso de la propiedad `ItemsSource` de la clase `TreeViewItem` puede resultar algo delicado. Impone el uso de un `ItemTemplate` para gestionar la jerarquía. El tipo del objeto que se provee es, en este caso, `HierarchicalDataTemplate`, que es un tipo de objeto derivado de la clase `DataTemplate`. La implementación de su uso se detalla paso a paso a continuación.

En primer lugar se crea el `TreeView`, así como la estructura de datos a partir de la que se alimentará el `TreeView`.

- MainWindow.xaml.cs:

```
public class RegionesDelMundo
{
    public string NombreRegión { get; set; }
    public List<País> ListaPaíses { get; set;  }
}

public class País
```

```
{
    public string Nombre { get; set; }
}

public partial class MainWindow : Window
{
    public List<RegiónDelMundo> Regiones { get; set;  }

    public MainWindow()
    {
        InitializeComponent();

        Regiones = new List<RegiónDelMundo>
        {
            new RegiónDelMundo
            {
                NombreRegión = "Europa",
                ListaPaíses = new List<País>
                {
                    new País { Nombre = "Reino Unido" },
                    new País { Nombre = "Dinamarca" },
                    new País { Nombre = "Suecia" }
                }
            }
        };


        this.DataContext = this;
    }
}
```

- Definición del `TreeView` en MainWindow.xaml:

```
<TreeView ItemsSource="{Binding Regiones}">
</TreeView>
```

Lo cual produce el siguiente resultado:

La clase `HierarchicalDataTemplate` hereda de `DataTemplate` y es posible especificar el `ItemTemplate` del `TreeView` como de este tipo.

```
<TreeView ItemsSource="{Binding Regiones}">
    <TreeView.ItemTemplate>
        <HierarchicalDataTemplate>
            <TextBlock Text="{Binding NombreRegión}" />
        </HierarchicalDataTemplate>
```

```
    </TreeView.ItemTemplate>
</TreeView>
```

El `TreeView` tiene ahora el siguiente aspecto:

La clase `HierarchicalDataTemplate` posee una propiedad `ItemsSource` que vincularemos con la lista de países:

```
...
<HierarchicalDataTemplate ItemsSource="{Binding ListaPaíses}">
...
```

Como con todos los `ItemsControl`, para obtener el resultado deseado, es necesario proveer un `DataTemplate` para los objetos de tipo `País`. La clase `HierarchicalDataTemplate` presenta una propiedad `ItemTemplate` que permite especificar el modelo de datos que se desea aplicar para estos objetos.

```
<TreeView ItemsSource="{Binding Regiones}">
    <TreeView.ItemTemplate>
        <HierarchicalDataTemplate ItemsSource="{Binding ListaPaíses}">
            <TextBlock Text="{Binding NombreRegión}" />
            <HierarchicalDataTemplate.ItemTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding NombrePaís}" />
                </DataTemplate>
            </HierarchicalDataTemplate.ItemTemplate>
        </HierarchicalDataTemplate>
    </TreeView.ItemTemplate>
</TreeView>
```

El resultado se corresponde con lo esperado:

g. Slider

Este control permite seleccionar un valor numérico comprendido entre dos extremos. El usuario realiza su selección

deslizando un cursor a lo largo de una línea. Este tipo de controles se utiliza, a menudo, en aplicaciones multimedia para gestionar el volumen del sonido o la situación de la lectura de un archivo de audio o de vídeo.

Los extremos mínimo y máximo se definen asignando valor a las propiedades `Minimum` y `Maximum`, mientras que el intervalo entre los valores seleccionables se define mediante la propiedad `SmallChange`. Si esta propiedad no posee ningún valor, es posible seleccionar cualquier valor decimal soportado por el tipo `double`. La propiedad `LargeChange` permite definir el número de unidades que debe recorrer el cursor hacia adelante o hacia atrás cuando el usuario haga clic en la línea, pero no deslice el cursor.

h. Calendar

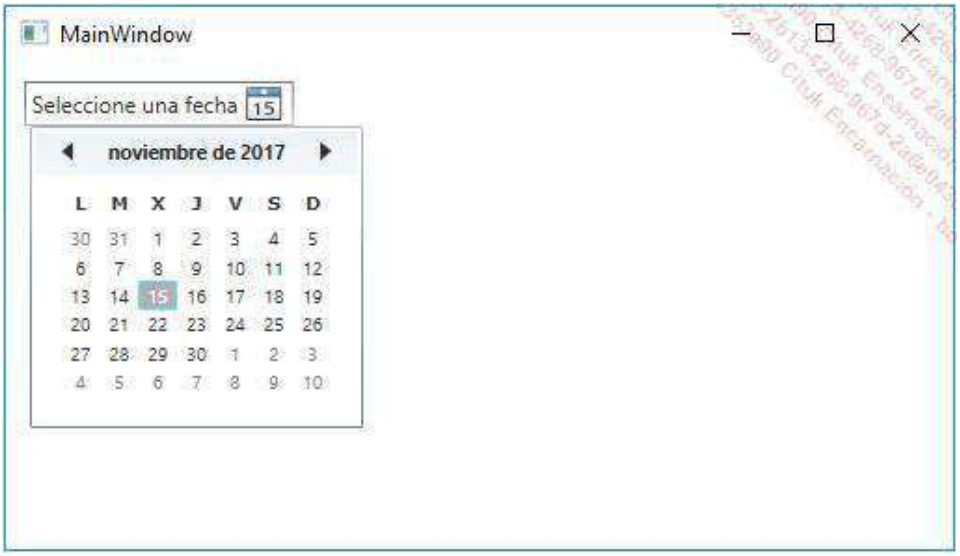
Este control permite navegar de manera visual en un calendario para seleccionar una fecha, de manera similar al calendario integrado en la barra de tareas de Windows.

La fecha seleccionada se representa mediante la propiedad `SelectedDate`.

i. DatePicker

La selección de una fecha resulta particularmente complicada de implementar, y por este motivo el framework .NET provee el control `DatePicker`. Este control está compuesto por tres partes: un campo que permite introducir texto, un calendario que puede estar visible u oculto y por último un botón que permite mostrar el calendario y, por tanto, introducir de manera directa una fecha y confirma que los datos que recibe se corresponden con una fecha válida. La selección de fecha mediante el calendario permite evitar cualquier problema relativo al formato de los datos y, a menudo, se da preferencia a su uso.

La propiedad `SelectedDate` permite acceder en lectura y escritura al valor seleccionado o introducido en el control.



6. Controles de acción

A diferencia de las aplicaciones de consola, cuyo ciclo de vida es muy simple, pues utilizan un modo de ejecución secuencial, las aplicaciones gráficas están a menudo compuestas por una multitud de módulos que ejecutan tareas específicas cuando el usuario lo desea. Para ejecutar estas tareas específicas, uno de los medios existentes es el uso de los controles de acción.

a. Button

El control `Button` es, sin duda, el más utilizado de los controles de acción. Su modo de funcionamiento es simple e intuitivo: se trata de un control cuya apariencia es la de un botón con un texto que indica su utilidad.

Con WPF, este control no permite solamente mostrar texto. En efecto, su propiedad `Content` es de tipo `Object` y es posible asignarle cualquier otro control de WPF: `Image`, `Grid` o `StackPanel` con contenido, texto o incluso un control personalizado.

Este control dispone, a su vez, de un elemento `Click` que permite ejecutar una sección de código como reacción al clic del usuario sobre el control.

```
<Button Content="¡Haga clic!" Click="Button_Click" />
```

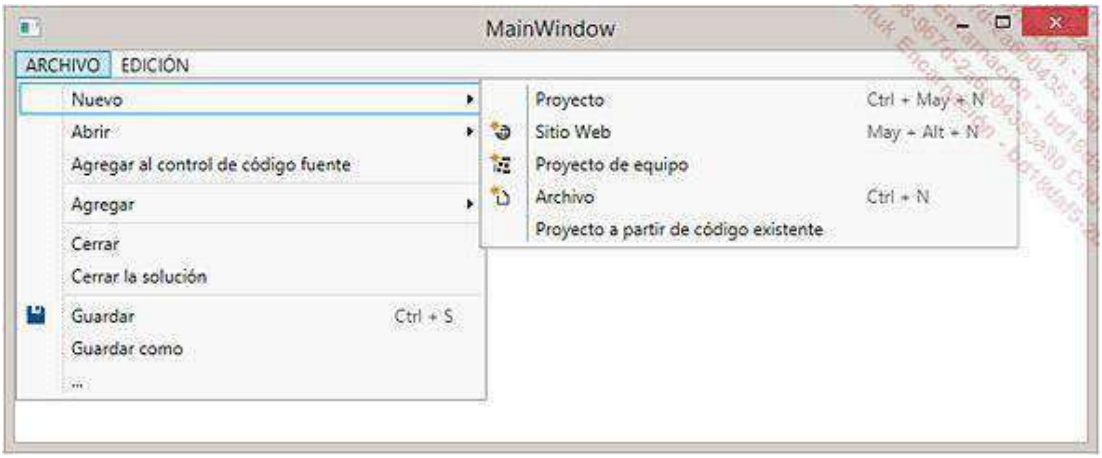
b. Menu

El control `Menu` se utiliza principalmente para presentar acciones comunes a toda la aplicación. Se representa de manera visual mediante una barra situada generalmente en la parte superior de la ventana principal de una aplicación. Es posible situar distintos controles `Menu`en las distintas ventanas que componen un programa.

Cada uno de los elementos que componen esta barra puede también contener otros elementos. Esta estructura es parecida a la de un `TreeView` y el código que permite su creación es, por tanto, muy similar al que permite definir un `TreeView` y su contenido. El control `Menu` contiene elementos `MenuItem` que pueden, a su vez, contener otros elementos de tipo `MenuItem`. El texto que indica la utilidad de cada uno de los botones recibe valor mediante la propiedad `Header` de cada objeto de tipo `MenuItem`. Es posible asignar una imagen pasando un objeto `Image` a la propiedad `Icon`, mientras que existe un texto que indica el atajo de teclado asociado al `MenuItem` mediante su propiedad `InputGestureText`.

Preste atención, `InputGestureText` es únicamente una propiedad de representación. El hecho de asignarle un valor no asocia la combinación de teclas con la ejecución de la acción relativa al `MenuItem`.

Para asociar una acción a un `MenuItem` es necesario asociar un controlador de eventos a su propiedad `Click` de la misma manera que con un control de tipo `Button`.



El menú anterior está generado a partir del código siguiente:

```
<Menu Height="20" VerticalAlignment="Top">
   <MenuItem Header="ARCHIVO">
      <MenuItem Header="Nuevo">
            <MenuItem Header="Proyecto" InputGestureText="Ctrl + May + N" />
            <MenuItem Header="Sitio Web" InputGestureText="May + Alt + N">
                <MenuItem.Icon>
                    <Image Source="NewWebSite_6288.png" />
                </MenuItem.Icon>
            </MenuItem>
            <MenuItem Header="Proyecto de equipo" >
                <MenuItem.Icon>
                    <Image Source="NewTeamProject_7437.png" />
                </MenuItem.Icon>
            </MenuItem>
            <MenuItem Header="Archivo" InputGestureText="Ctrl + N">
                <MenuItem.Icon>
                    <Image Source="NewFile_6276.png" />
                </MenuItem.Icon>
            </MenuItem>
            <MenuItem Header="Proyecto a partir de código existente" />
      </MenuItem>
      <MenuItem Header="Abrir">
            <MenuItem Header="Proyecto/Solución"
InputGestureText="Ctrl + Maj + A" />
            <MenuItem Header="Sitio Web" InputGestureText="Maj + Alt + A" />
            <Separator />
            <MenuItem Header="Abrir desde el control de código
fuente" InputGestureText="Maj + Alt + N" />
            <MenuItem Header="Proyecto de equipo" />
            <Separator />
```

```
            <MenuItem Header="Archivo" InputGestureText="Ctrl + A" />
            <MenuItem Header="Convertir" />
      </MenuItem>
      <MenuItem Header="Agregar al control de código fuente" />
      <Separator />
      <MenuItem Header="Agregar">
            <MenuItem Header="Nuevo proyecto"  />
            <MenuItem Header="Nuevo sitio Web"  /
            <Separator />
            <MenuItem Header="Proyecto existente" />
            <MenuItem Header="Sitio Web existente" />
      </MenuItem>
      <Separator />
      <MenuItem Header="Cerrar" />
      <MenuItem Header="Cerrar la solución" />
      <Separator />
      <MenuItem Header="Guardar" InputGestureText="Ctrl + S">
            <MenuItem.Icon>
                <Image Source="Save_6530.png" />
            </MenuItem.Icon>
      </MenuItem>
      <MenuItem Header="Guardar como" />

      <MenuItem Header="..." />
    </MenuItem>
    <MenuItem Header="EDICIÓN">
      <MenuItem Header="Deshacer" InputGestureText="Ctrl + Z" />
      <MenuItem Header="Rehacer" InputGestureText="Ctrl + Y" />
      <MenuItem Header="Deshacer la última acción global" />
      <MenuItem Header="Rehacer la última acción global" />

      <MenuItem Header="..." />
   </MenuItem>
</Menu>
```

Las imágenes utilizadas para realizar este menú provienen de la **Biblioteca de imágenes de Visual Studio**, disponible en la siguiente dirección: https://www.microsoft.com/en-us/download/details.aspx?id=35825

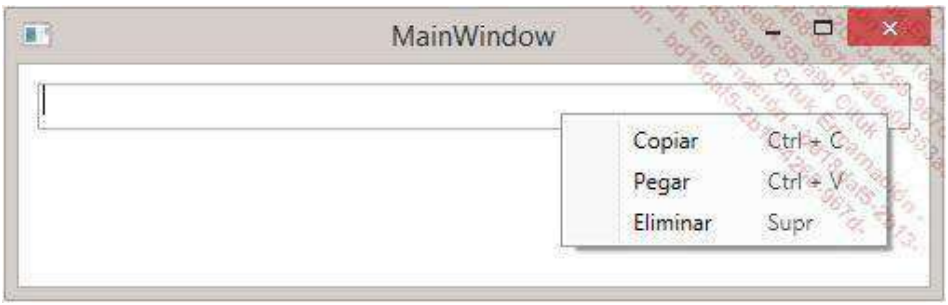
Como con el objeto `TreeView`, es posible construir un `Menu` asignando valor a su propiedad `ItemsSource` y proporcionando un `ItemTemplate` de tipo `HierarchicalDataTemplate`.

c. ContextMenu

La creación de un menú contextual se realiza en WPF mediante un control `ContextMenu`. Este se concibe sobre la misma base que el control `Menu`: ambos heredan de la clase `MenuBase`. El diseño y uso de ambos controles resulta muy similar. El objeto `ContextMenu` también contiene elementos de tipo `MenuItem` que es posible anidar. Las principales diferencias entre ambos controles son que el menú contextual está vinculado a un control mediante la propiedad `ContextMenu` del control y que aparece bajo demanda cuando el usuario hace clic con el botón derecho del ratón.

El siguiente código permite crear un control `TextBox` y le asigna un menú contextual.

```
 <TextBox Height="25" VerticalAlignment="Top"  Margin="10">
    <TextBox.ContextMenu>
        <ContextMenu>
           <MenuItem Header="Copiar" InputGestureText="Ctrl + C" />
           <MenuItem Header="Pegar" InputGestureText="Ctrl + V" />
           <MenuItem Header="Eliminar" InputGestureText="Supr" />
        </ContextMenu>
    </TextBox.ContextMenu>
</TextBox>
```



d. ToolBar

Los controles `ToolBar` y `ToolBarTray` permiten crear una barra de herramientas completa para una ventana.

Cada control `ToolBar` forma un grupo de controles que proporcionan funcionalidades similares o ligadas entre sí. El contenedor dedicado a los controles de tipo `ToolBar` se representa mediante el tipo `ToolBarTray`. Este gestiona las operaciones de disposición, de redimensionamiento y de desplazar-soltar de los elementos `ToolBar`.

Las distintas `Toolbar` se ubican en su contenedor mediante las propiedades `Band` y `BandIndex` que representan, respectivamente, el índice de la barra en la que se debe ubicar el control y la ubicación en la barra respecto a las demás `Toolbar`.

Un control `ToolBar` puede contener cualquier tipo de control: botones, listas desplegables, opciones a marcar, etc.

El siguiente código permite generar una barra de herramientas completa repartida en dos barras.

```
<ToolBarTray  VerticalAlignment="Top">
    <ToolBar Band="1" BandIndex="1">
        <Button>
            <Image Source="Cut_6523.png" />
        </Button>
        <Button>
            <Image Source="Copy_6524.png" />
        </Button>
        <Button>
            <Image Source="Paste_6520.png" />
        </Button>
        <Separator />
        <Button>
            <Image Source="Undo_16x.png" />
```

```
        </Button>
        <Button>
            <Image Source="Redo_16x.png" />
        </Button>
    </ToolBar>
    <ToolBar Band="2" BandIndex="1">
        <Button>
            <Image Source="Save_6530.png" />
        </Button>
    </ToolBar>
</ToolBarTray>
```

Interacciones de teclado y de ratón

En un entorno gráfico como el que proporcionan las aplicaciones WPF es esencial ser capaz de reaccionar a las interacciones entre el usuario y la aplicación para proveer una experiencia fluida y coherente. En la actualidad, estas interacciones se realizan casi por completo con la ayuda de dos dispositivos: el teclado y el ratón. WPF proporciona distintos eventos que se producen cuando el usuario realiza alguna acción sobre uno de estos dispositivos. Estos eventos se generan mediante el tipo `System.Windows.UIElement`, que es un ancestro común a casi la totalidad de los controles WPF.

1. Eventos de teclado

Con WPF existen dos eventos principales producidos cuando el usuario presiona una tecla de su teclado: `KeyDown` y `KeyUp`. Se corresponden, respectivamente, con la pulsación de una tecla y con la liberación de la misma y se producen en este orden.

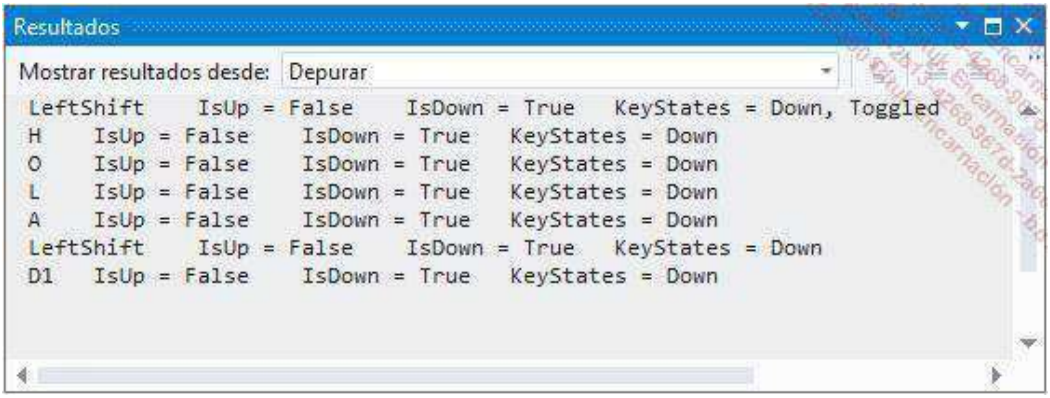
El delegado asociado a ambos posee la siguiente definición:

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e);
```

Los datos que se transmiten en el parámetro de tipo `KeyEventArgs` permiten conocer la tecla física del teclado relacionada con el evento producido, así como su estado. El siguiente ejemplo de código muestra en la ventana **Salida** de Visual Studio el nombre de la tecla manipulada así como su estado mediante las propiedades `IsUp`, `IsDown` y `KeyStates` del objeto `KeyEventArgs`.

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        KeyDown += Window_ManipulacionTeclaTeclado;
    }

    private void Window_ManipulacionTeclaTeclado(object sender,
KeyEventArgs e)
    {
        Console.WriteLine("{0} \t IsUp = {1} \t IsDown = {2} \t
KeyStates = {3}", e.Key, e.IsUp, e.IsDown,  e.KeyStates);
    }
}
```

De este modo, cuando la ventana de la aplicación está activa, el hecho de escribir `Hola!` muestra el siguiente resultado en la ventana **Resultados**.



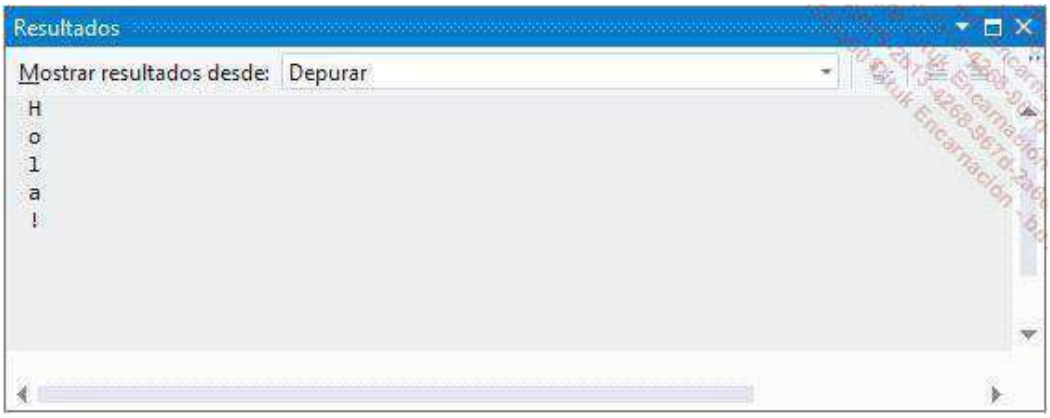
La tecla `LeftShift` se utiliza para escribir la primera letra en mayúscula. La tecla `D1`, que se encuentra en la última posición, se corresponde con el signo de exclamación.

Es posible obtener el texto tecleado carácter por carácter utilizando el evento `TextInput`. El delegado asociado es el siguiente:

```
public delegate void TextCompositionEventHandler(object sender,
TextCompositionEventArgs e);
```

El parámetro de tipo `TextCompositionEventArgs` posee una propiedad `Text` que contiene el texto asociado a una pulsación sobre alguna tecla, si ésta genera texto.

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        TextInput += MainWindow_TextInput;
    }

    void MainWindow_TextInput(object sender,
TextCompositionEventArgs e)
    {
        Console.WriteLine(e.Text);
    }
}
```



2. Eventos de ratón

El ratón genera muchos más eventos distintos que el teclado. Es posible pulsar sobre alguno de sus botones, mover su rueda o desplazarlo por la pantalla. De este modo, sobre cada control existe aproximadamente una decena de eventos dedicados a la gestión de las interacciones mediante el ratón. Su nombre empieza, sistemáticamente, por `Mouse`.

MouseDown y MouseUp

Estos dos eventos se producen cuando se pulsa o se libera algún botón del ratón. El botón correspondiente puede ser el izquierdo, el derecho, un botón situado en el centro que generalmente es la rueda, o incluso un botón extendido, es decir, que no se corresponde funcionalmente con un ratón clásico. Los botones extendidos, cuando están presentes, se sitúan generalmente a un lado.

El parámetro `MouseButtonEventArgs` que se pasa a los controladores de estos eventos posee una propiedad `ChangedButton` que indica el botón que se ha manipulado. Provee, a su vez, una propiedad `ButtonState` que indica si se ha pulsado o no el botón y una propiedad `ClickCount` que permite saber si se han realizado varios clics. Si deseara implementar una funcionalidad de gestión del triple clic, esta última propiedad le ayudará.

MouseLeftButtonDown y MouseLeftButtonUp

Estos eventos se producen específicamente cuando se manipula el botón izquierdo del ratón. Su funcionamiento es idéntico al de `MouseDown` y `MouseUp`.

MouseRightButtonDown y MouseRightButtonUp

Estos eventos se producen específicamente cuando se manipula el botón derecho del ratón. Su funcionamiento es idéntico al de `MouseDown` y `Mouse Up`.

MouseMove

Este evento se produce con cada movimiento del cursor del ratón sobre el control que gestiona el evento. Los datos que se pasan al controlador del evento no son relativos a la posición del ratón. Para conocer las coordenadas del cursor es preciso utilizar el método `GetPosition` de la clase estática `Mouse`. Este método recibe como parámetro un objeto que implementa la interfaz `IInputElement`: es posible pasar a este método cualquier control WPF.

En el archivo `.xaml`:

```
<TextBox x:Name="txtNombre" />
```

En el archivo `.xaml.cs`:

```
double posicionHorizontal = Mouse.GetPosition(txtNombre).X
double posicionVertical =  Mouse.GetPosition(txtNombre).Y
```

MouseEnter y MouseLeave

Estos eventos se producen cuando el cursor entra o sale de un control. Para estos dos eventos, el controlador recibe un parámetro de tipo `MouseEventArgs`. Este permite analizar el estado actual del ratón mediante sus propiedades `LeftButton`, `RightButton`, `MiddleButton`, `X1Button` y `X2Button`. Estas propiedades proveen el estado de los distintos botones del dispositivo. Estos eventos pueden resultar particularmente interesantes si desea implementar la totalidad del código necesario para implementar la funcionalidad de arrastrar y colocar.

3. Arrastrar y colocar

Cuando se trabaja con WPF, la gestión de la funcionalidad arrastrar y colocar se simplifica gracias a distintos elementos:

- Cada control puede ser una fuente o un destino de la funcionalidad arrastrar y colocar.
- El método estático `DoDragDrop` de la clase `DragDrop` implementa la capacidad de almacenar temporalmente datos desplazados y se ocupa también de la gestión de los efectos visuales asociados a la operación.
- Existen varios eventos que permiten proveer los procesamientos y la experiencia visual al usuario en función de la evolución de la operación.

La implementación de esta acción se realiza mediante un ejemplo sencillo: el desplazamiento de un disco rojo sobre un rectángulo verde hace que el color del rectángulo cambie a rojo.

- En primer lugar, cree un nuevo objeto de aplicación WPF llamado `ArrastrarColocar`.
- Modifique el código del archivo `MainWindow.xaml` para situar un control `Ellipse` (elipse) y un control `Rectangle` (rectángulo) en él.

```
<Window x:Class="ArrastrarColocar.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="468.4" Width="553.8">
    <Canvas>
        <Ellipse Canvas.Left="50" Canvas.Top="50" Height="100"
                 Width="100" Fill="#FF0000" />

        <Rectangle Height="100" Width="100" Canvas.Left="400"
                   Canvas.Top="280" Stroke="#000000"
                   Fill="#00FF00" StrokeThickness="3"
    </Canvas>
</Window>
```

La estructura visual está implementada. A partir de ahora, conviene tener en cuenta tres elementos:

- El control `Ellipse` es el origen de la acción arrastrar y colocar.
- El `Rectangle` es el destino de dicha acción.
- Se transmite un dato desde la `Ellipse` hacia el `Rectangle`: el color.

Comenzamos autorizando al `Rectangle` a ser el destino de una operación de arrastrar y colocar.

Agregue el atributo `AllowDrop` al `Rectangle` y asigne el valor `True`.

```
<Rectangle .... AllowDrop="True" />
```

A continuación es necesario gestionar la producción de la acción. Para ello, controlaremos el movimiento del ratón sobre el control `Ellipse` mientras esté pulsado el botón izquierdo del ratón. El evento `MouseMove` es ideal para gestionar esta situación.

Suscríbase al evento `MouseMove` del control `Ellipse` y proporcione el siguiente controlador de eventos:

```
private void Ellipse_MouseMove(object sender, MouseEventArgs e)
{
    if (e.LeftButton == MouseButtonState.Pressed)
    {
        var ellipse = (Ellipse)sender;
        DragDrop.DoDragDrop(ellipse,  ellipse.Fill.ToString(),
DragDropEffects.Copy);
    }
}
```

Este controlador produce la operación de arrastrar sobre el control `Ellipse`, almacena el valor del color de relleno de dicho control de manera que pueda reutilizarse más adelante y, por último, aplica el efecto visual "Copia". Este efecto modifica el cursor cuando se alcanza un destino, como el control `Rectangle`.

Una vez que podemos arrastrar nuestra `Ellipse`, queda gestionar la parte "colocar" de la operación.

Para ello, suscríbase al evento `Drop` del `Rectangle` y cree su controlador de eventos.

```
private void Rectangle_Drop(object sender, DragEventArgs e)
{
    Rectangle rectangle = sender as Rectangle;
    if (rectangle != null)
    {

       //Se comprueba la existencia de un dato almacenado
       //para la acción de arrastrar y colocar

       if (e.Data.GetDataPresent(DataFormats.StringFormat))
       {
```

```
            //Se recupera el dato. Se trata de una cadena de
            //caracteres cuyo valor es  "#FFFF0000"

            string dataString =
(string)e.Data.GetData(DataFormats.StringFormat);

            //Se utiliza un objeto BrushConverter para transformar
            //esta cadena de caracteres en Brush aplicable
            //a la propiedad Fill del rectángulo

            BrushConverter converter = new BrushConverter();
            if (converter.IsValid(dataString))
            {
                Brush newFill =
(Brush)converter.ConvertFromString(dataString);
                rectangle.Fill = newFill;
            }
        }
    }
}
```

Tras el evento `Drop`, se recupera el dato almacenado previamente mediante el método `DoDragDrop`. Se transforma dicho dato de manera que pueda utilizarse: aquí, se utiliza un `BrushConverter` cuyo objetivo es convertir una cadena de caracteres en un objeto `Brush` (pincel) que puede utilizarse para dar color a un elemento. Se aplica, por último, el valor objetivo a la propiedad `Fill` del control `Rectangle`.

- Ejecute la aplicación y realice la operación de arrastrar y colocar desde el círculo hacia el rectángulo: ¡comprobará que el rectángulo cambia de color!

Los eventos `DragEnter` y `DragLeave` permiten agregar código entre el inicio de la operación y su final. `DragEnter` se produce cuando un control se arrastra sobre otro control que autoriza esta operación, sin colocarlo. `DragLeave` se ejecuta cuando el control se arrastra fuera de los límites de otro control. Estos dos eventos ofrecen, de este modo, la capacidad, entre otras, de realizar una previsualización del efecto correspondiente a la acción de arrastrar y colocar.

Ir más allá con WPF

WPF es una tecnología cuya estructura permite llevar muy lejos la personalización de una interfaz gráfica. Visual Studio cuenta con muchas herramientas, entre ellas **Blend**, que resultan de gran ayuda para la elaboración de interfaces gráficas.

WPF permite de este modo, gracias a los **bindings** en particular, un **desacople** casi perfecto entre la parte visual de una aplicación y el código asociado a ella. La implementación de esta separación se realiza, a menudo, mediante el patrón de diseño **MVVM** (*Model - View - ViewModel*).

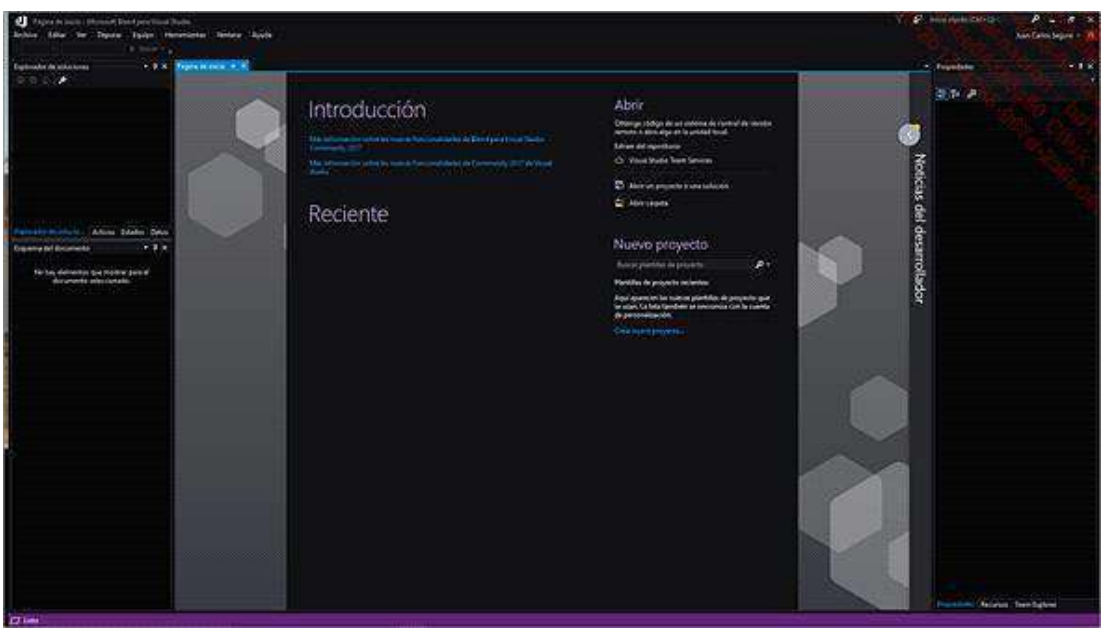
1. Introducción al uso de Blend

Blend para Visual Studio es una herramienta que aparece, inicialmente, bajo el nombre de **Expression Blend** en la suite Expression de Microsoft en 2007. Tras sus inicios, se destina a producir código XAML para aplicaciones **WPF** y su radio de acción se amplia para incluir **Silverlight** y **Windows Phone 7**. Tras la versión 4, se ha introducido en la gama de herramientas proporcionadas con Visual Studio, y el conjunto de tecnologías que soporta ha evolucionado incluyendo aplicaciones **Windows Phone** y **Windows 8/8.1**, y posteriormente las aplicaciones universales para las plataformas **Windows 10 (PC y smartphones)**. Actualmente, Blend puede utilizarse para crear interfaces de aplicaciones Windows Phone o Windows Store en **HTML5**.

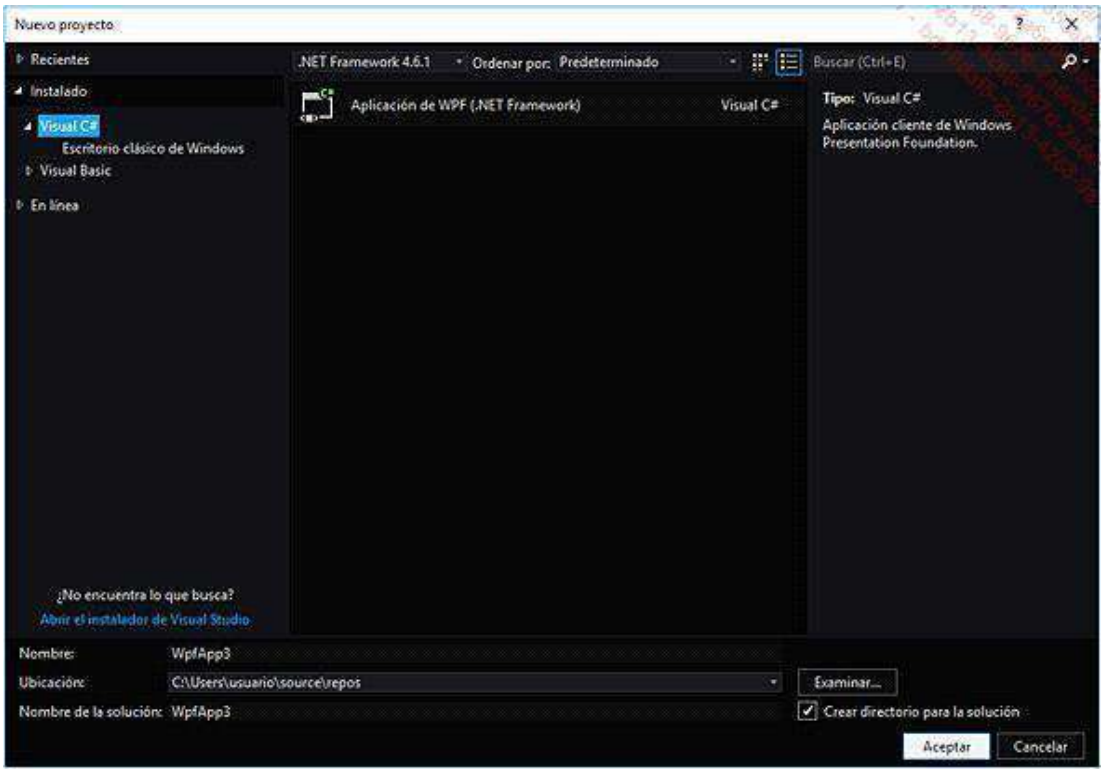
Blend puede describirse como una versión orientada al diseño de Visual Studio. En efecto, Visual Studio posee un diseñador visual básico y decenas de herramientas destinadas a guiar al desarrollador en su tarea de escribir código. Blend, por su parte, está formado por un editor de código así como un diseñador visual complementado por numerosas herramientas gráficas. La brecha entre Visual Studio y Blend aumenta con la aparición de cada nueva versión, la última incluye el soporte de IntelliSense y una interfaz muy cercana a la de Visual Studio, lo cual favorece una curva de aprendizaje muy rápida.

a. La interfaz

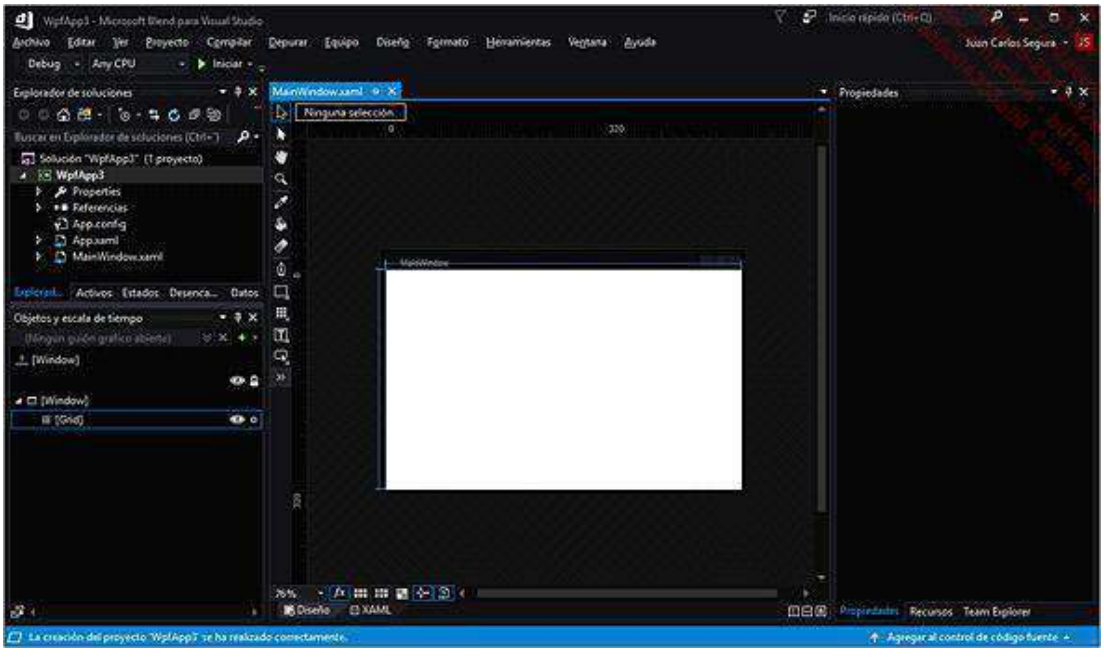
La interfaz general de Blend 2017 es idéntica a la interfaz de Visual Studio, con la única diferencia de que el tema por defecto para esta herramienta es específico de Blend y es sobre todo negro para poder visualizar mejor la interfaz gráfica durante su edición.



La opción **Crear nuevo proyecto...** en la pantalla de inicio abre la siguiente ventana, que permite seleccionar el tipo de proyecto sobre el que desea trabajar. La lista de plantillas depende de las funcionalidades instaladas con Visual Studio. Para una instalación destinada solo al desarrollo de aplicaciones de escritorio clásicas, tiene el siguiente aspecto:



La opción **Aplicación WPF** presenta la siguiente pantalla, cuya organización debería recordarle a Visual Studio. Presenta por defecto tres zonas repartidas en la parte izquierda, la sección central y la parte derecha.

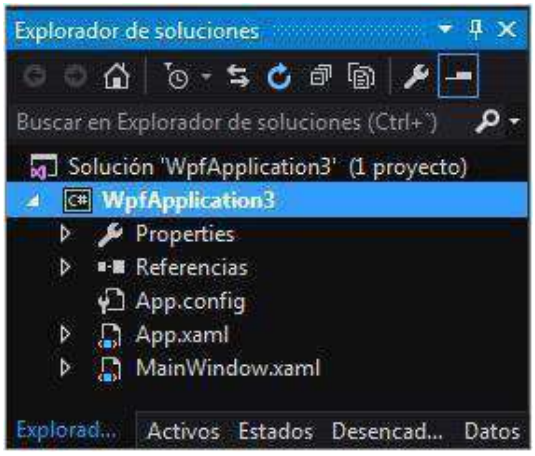


En la sección central encontramos la superficie de trabajo, que representa la ventana en curso de desarrollo. Es posible realizar un zoom fácilmente mediante la rueda del ratón para visualizar mejor los elementos que se sitúan en esta zona.

Diversas ventanas ancladas a derecha e izquierda contienen herramientas que permiten visualizar y modificar ciertos elementos de la aplicación.

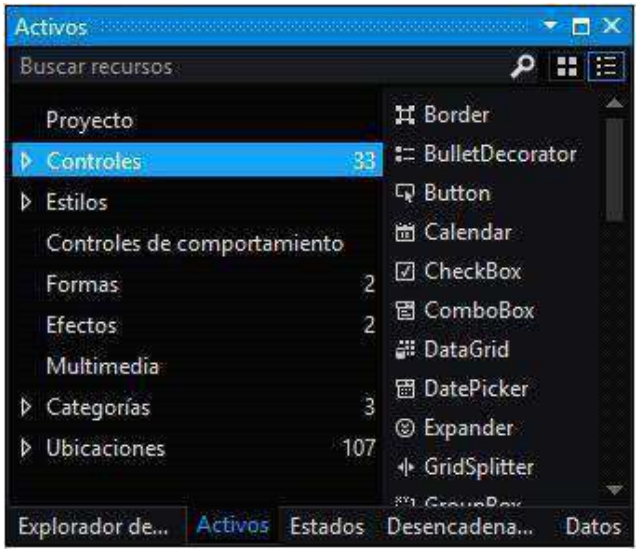
Explorador de soluciones

Como en Visual Studio, esta herramienta contiene un árbol que representa la estructura del proyecto en curso de edición. Permite, entre otros, navegar en los proyectos y abrir los archivos que contienen.



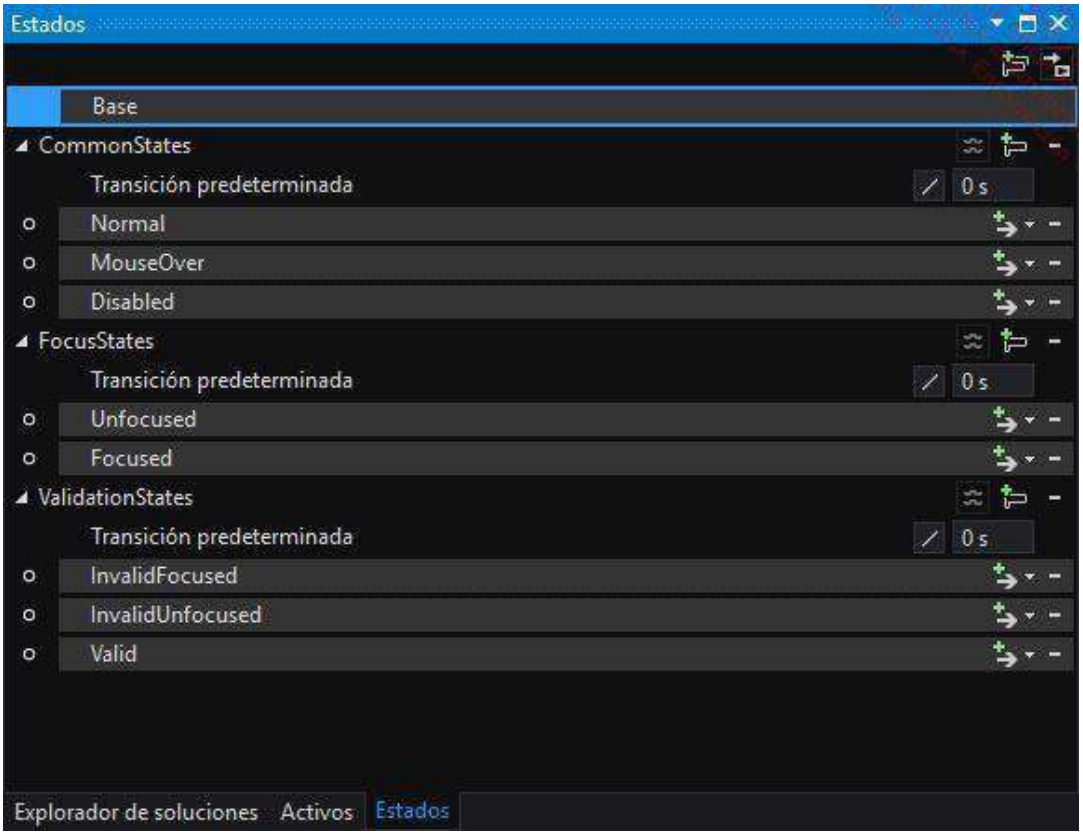
Activos

Los distintos elementos enumerados en esta pestaña son los controles (principalmente gráficos), estilos visuales, comportamientos o elementos gráficos (imágenes, vídeos, etc.) que pueden utilizarse en aplicaciones WPF. Están agrupados en distintas categorías que representan ubicaciones físicas o agrupaciones lógicas.



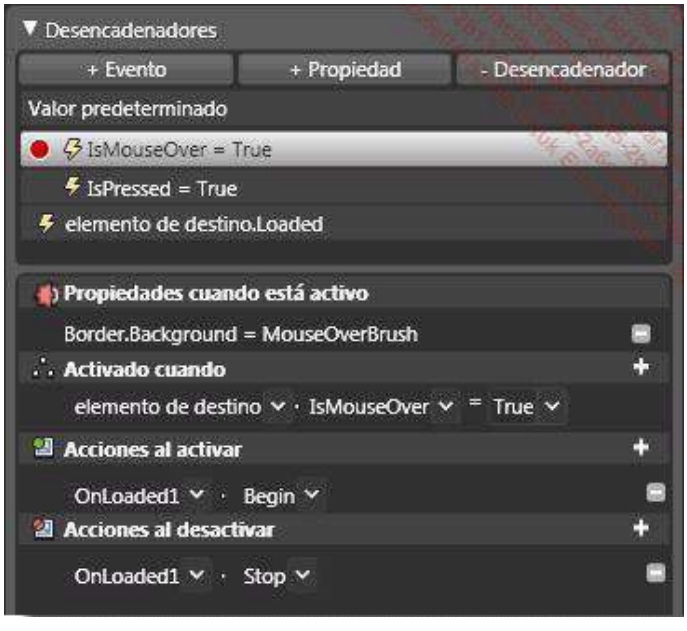
Estados

Los controles gráficos pueden tener distintos estados visuales definidos en XAML. Cada uno agrupa un conjunto de valores de propiedades que deben aplicarse cuando se modifica el estado visual del control.



Desencadenadores

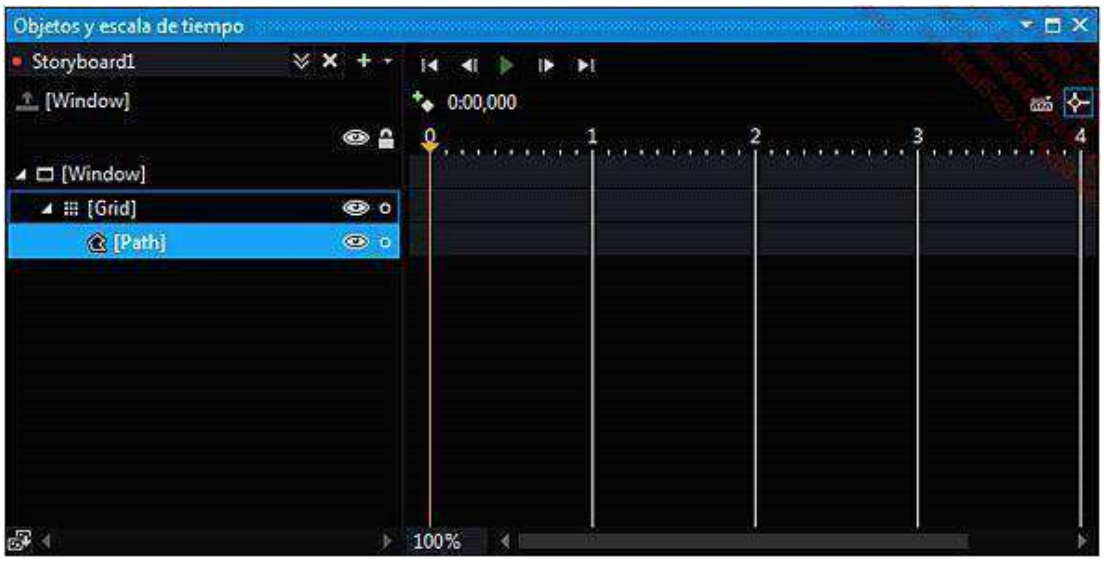
Con XAML es posible declarar modificaciones que deben aplicarse cuando se cumplen una o varias condiciones. Estas modificaciones se agrupan en los desencadenadores (o *triggers*). La pestaña **Desencadenadores** permite crear y gestionar estos objetos sin utilizar directamente XAML.



Objetos y escala de tiempo

Esta pestaña tiene dos objetivos: visualizar el árbol lógico formado por los controles de una ventana y generar

escenarios para las animaciones (comúnmente llamados *storyboards*). Las animaciones pueden modificar las propiedades de varios controles, esta pestaña permite visualizar estos cambios mediante una escala de tiempo (diagrama que sigue una línea temporal).

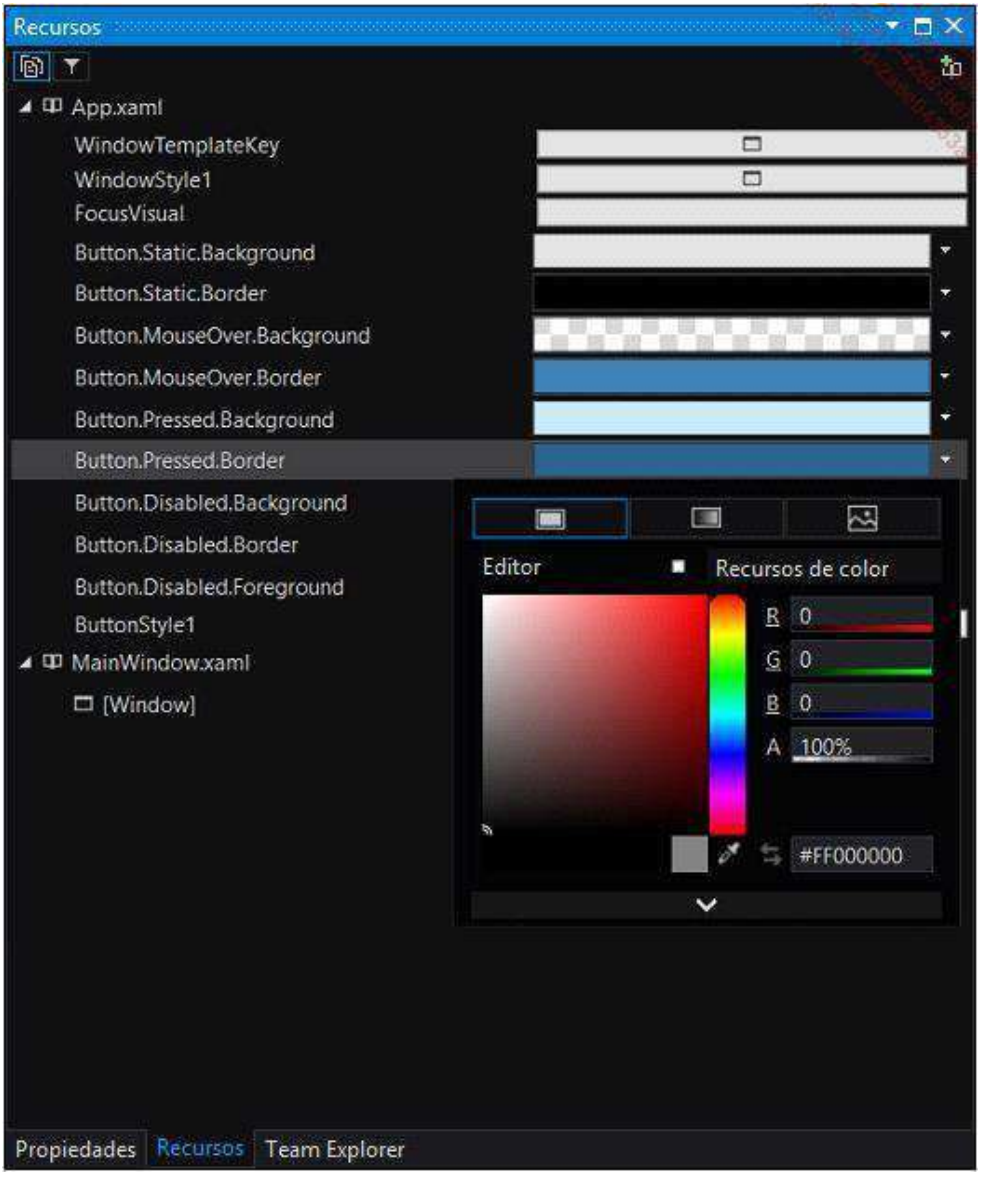


Propiedades

Esta pestaña permite modificar directamente las propiedades del control seleccionado en la pestaña **Objetos y escala de tiempo**. Es similar a la pestaña **Propiedades** disponible en Visual Studio, pero contiene funcionalidades suplementarias como la selección de un recurso en una lista de entre todos los recursos accesibles.

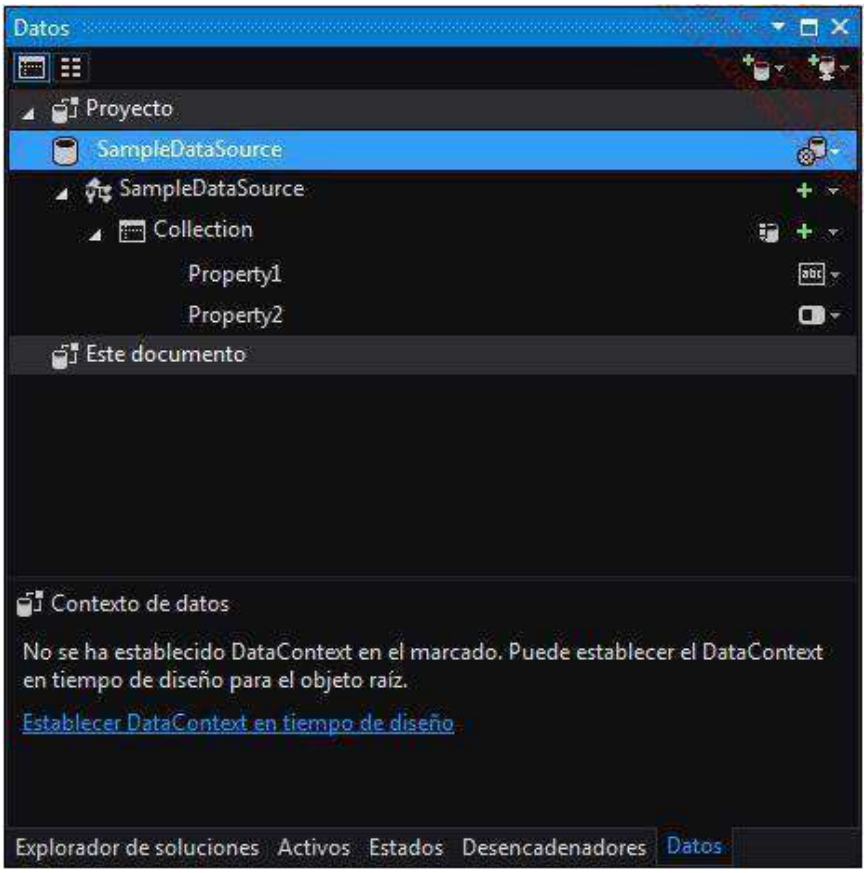
Recursos

Esta ventana permite enumerar los distintos recursos XAML del proyecto en curso, ya sean gráficos, textuales, numéricos o de cualquier otro tipo. Es posible, si su tipo está soportado por Blend, editarlos directamente.



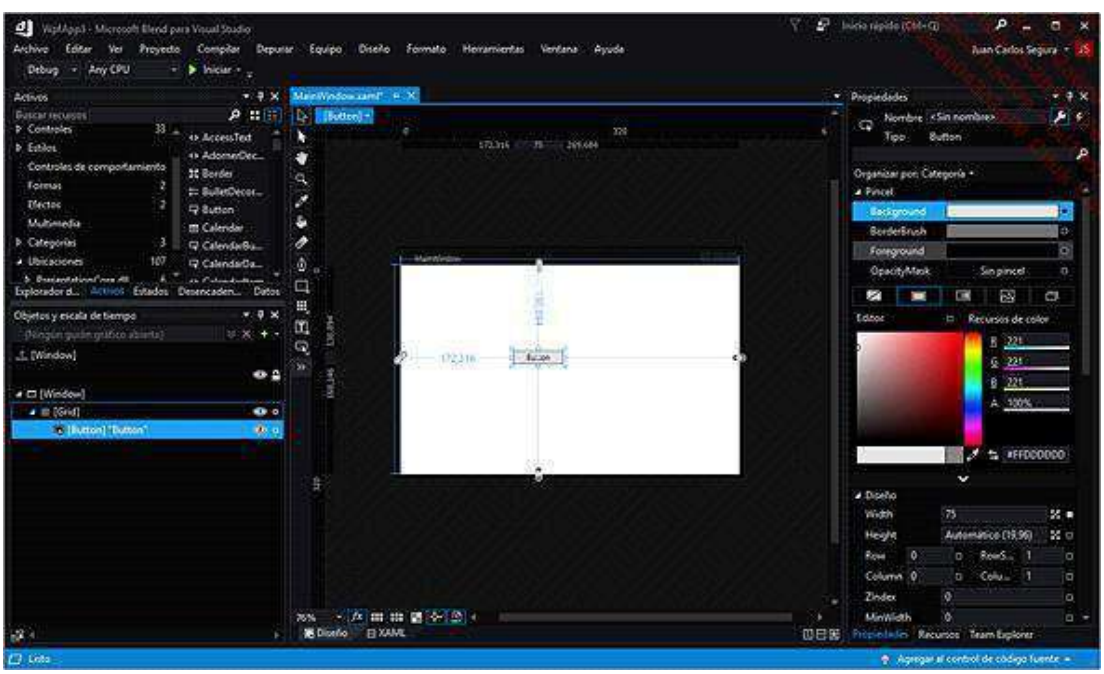
Datos

Esta última pestaña permite crear muestras de datos que se pueden utilizar en el diseño visual de la aplicación. Los controles pueden completarse con datos para que no sea necesario ejecutar la aplicación para verificar el efecto que produciría una modificación.



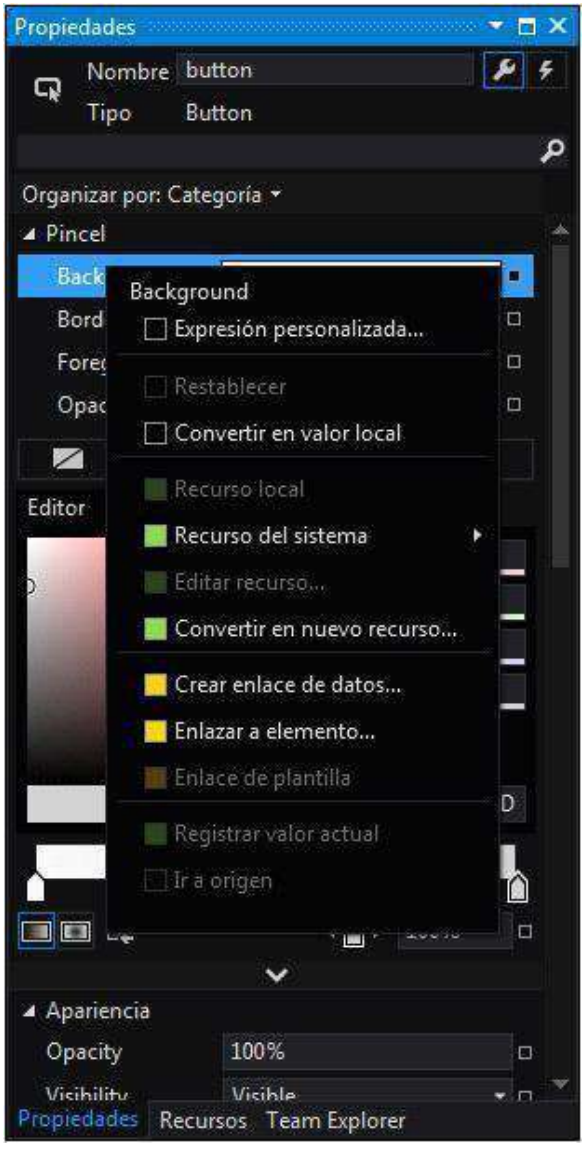
b. Agregar y modificar controles visuales

Es posible agregar controles a la superficie de diseño mediante una acción de arrastrar y colocar desde la pestaña **Activos** hacia el lugar de destino. Una vez creado, el control aparece en la pestaña **Objetos y escala de tiempo** y es posible seleccionarlo.

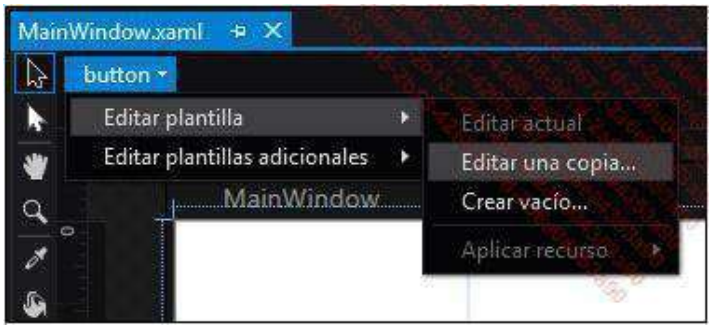


Cuando se selecciona el control, la pestaña **Propiedades** enumera las propiedades del control y permite editarlo. De la misma manera, las pestañas **Estados** y **Desencadenadores** se actualizan para reflejar los estados

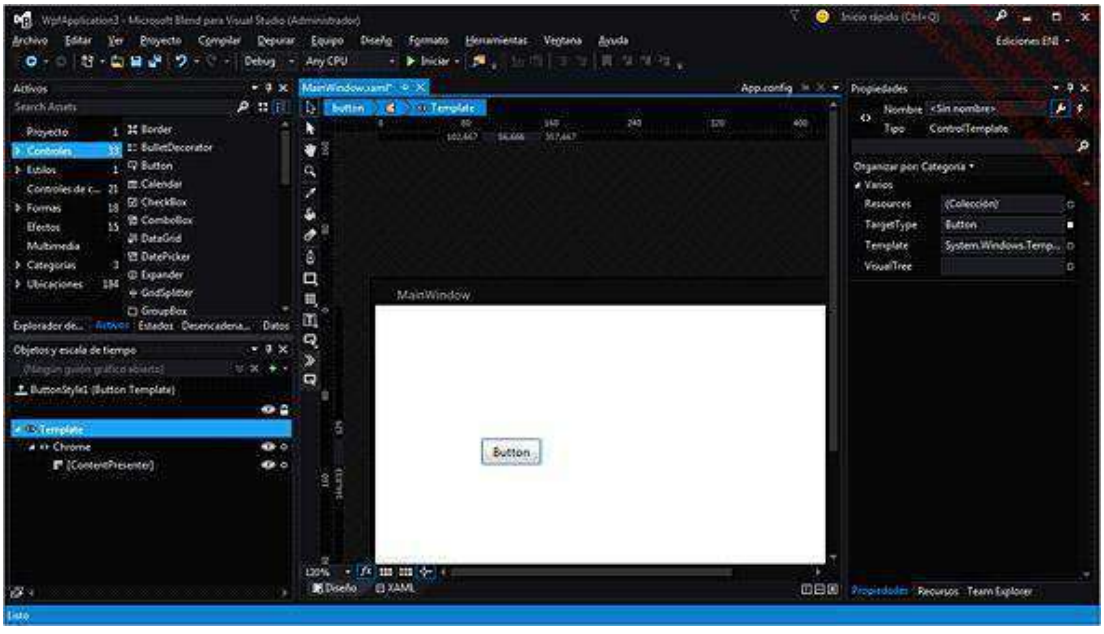
existentes en el control y los desencadenadores que tiene asociados.



Cuando el control seleccionado posee una plantilla, es posible editarla mediante un menú situado encima de la superficie de diseño.



Seleccionando la opción **Editar plantilla - Editar una copia** se modifica el árbol lógico de la pestaña **Objetos y escala de tiempo**: Blend se pone en modo "edición de plantilla", y las distintas pestañas que componen la interfaz visual se actualizan para reflejar los datos de la plantilla del control.



El uso de Blend como herramienta de edición gráfica permite ahorrar un tiempo considerable cuando se domina bien. Es necesario algo de tiempo para aprender cómo editar los distintos elementos que componen una interfaz WPF con la ayuda de Blend. Esta herramienta es un verdadero aliado del desarrollador, y si bien aquí hemos pasado por encima de lo que sería su uso más básico, sin duda puede imaginarse todo su potencial.

2. Introducción a MVVM

MVVM (*Model - View - ViewModel*) es un patrón de diseño que facilita la separación de la interfaz gráfica y del código que contiene la lógica de la aplicación. Aparecido con la evolución de WPF es, a día de hoy, indisociable de la noción de **binding**. Gracias a la existencia de estos vínculos de datos es posible realizar una separación eficaz entre las vistas y los datos que muestran.

a. Presentación

Este modelo de desarrollo define una estructura de aplicación desacoplada en tres capas distintas: `View`, `Model` y `ViewModel`.

View

Esta capa contiene el código relativo a la interfaz gráfica. Está compuesta por archivos de código XAML y archivos de code-behind C# asociados. En esta capa de la aplicación no debería manipularse ningún dato salvo el específico de la interfaz. Está vinculada con la lógica de la aplicación mediante los bindings.

Model

En esta capa podemos encontrar la definición y manipulación de los datos de la aplicación. Los procesamientos puramente definidos por el negocio de la aplicación se encuentran todos en esta capa.

ViewModel

Un `ViewModel` (modelo de vista) es una adaptación de datos propios de la capa `Model` con el objetivo de representarlos mediante la capa `View`. Podemos encontrar aquí la definición de los comandos utilizados por la

interfaz gráfica.

b. Las interfaces INotifyPropertyChanged e INotifyCollectionChanged

WPF permite vincular una o varias propiedades de un control a datos de un `ViewModel` mediante expresiones binding.

Se evalúan, inicialmente, al finalizar la carga de la ventana. De este modo, si alguna propiedad del `ViewModel` recibe valor durante esta carga, su valor se pasará también a los objetos que tenga vinculados. Pero si este valor se modifica mediante la interfaz de código C# tras la primera evaluación de la expresión de binding, la modificación no se repercutirá.

Para que el comportamiento de los bindings sea constante es necesario que el `ViewModel` implemente la interfaz `INotifyPropertyChanged`. En efecto, WPF utiliza el evento `PropertyChanged` de los objetos que implementan esta interfaz para saber que se ha producido alguna modificación y, si fuera necesario, repercutirla. El evento `PropertyChanged` debe producirse, evidentemente, a partir del `ViewModel` cuando se modifique el valor de alguna propiedad.

El caso de las colecciones es ligeramente diferente. Los elementos contenidos en una colección pueden modificarse sin que el valor intrínseco de la colección cambie. En este caso, el uso de la interfaz `INotifyPropertyChanged` no resuelve el problema: es la interfaz `INotifyCollectionChanged` la que recibe el testigo para indicar una modificación del elemento "hijo". Las colecciones que implementan esta interfaz producen un evento `CollectionChanged` cuando se produce un cambio sobre alguno de sus elementos. El motor de WPF intercepta este evento y propaga la información a los controles vinculados a la colección.


Solo existe una colección que implementa esta interfaz disponible de manera nativa en el framework .NET: `ObservableCollection<T>`.

c. Comandos

Un comando representa una acción que puede proporcionarse a ciertos controles mediante una expresión de binding. Se trata de la encapsulación de uno o dos delegados. El primero define el procesamiento que debe realizarse una vez se produzca la acción. El segundo, opcional, define una función que devuelve un valor booleano. Este valor indica si es posible realizar la acción en el contexto en curso.

Los controles `Button` y `MenuItem` permiten, en particular, utilizar comandos mediante su propiedad `Command`. En ambos casos, el comando se ejecuta cuando el usuario hace clic sobre el control.

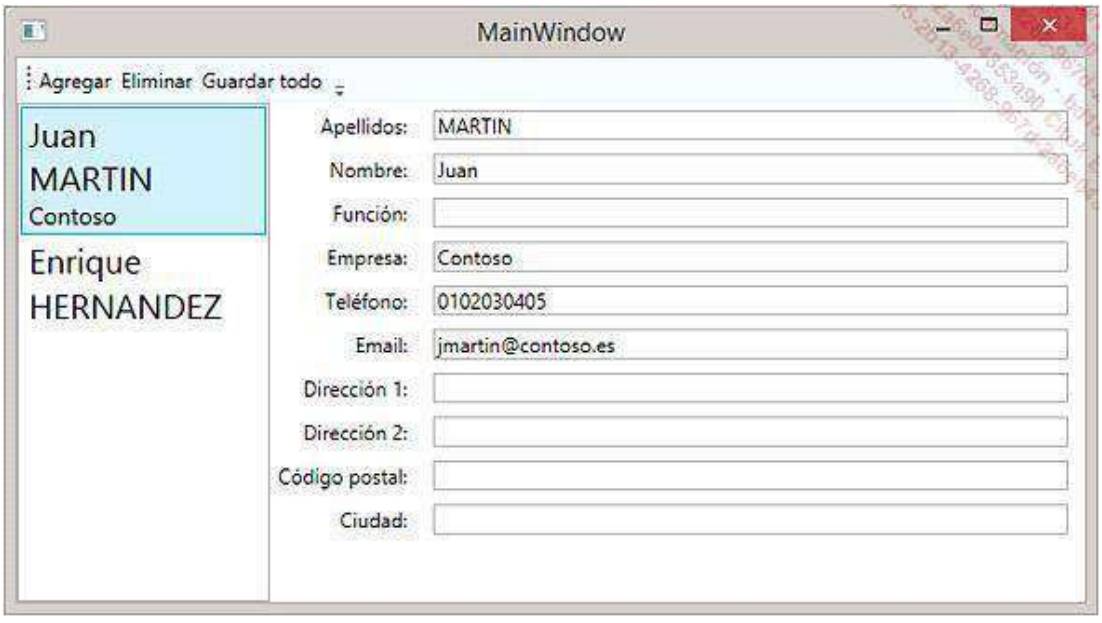
Los comandos se utilizan a menudo en contextos MVVM, pues permiten configurar fácilmente procesamientos no vinculados a la interfaz fuera de la capa `View`.

d. Implementación

Para estudiar la implementación de los conceptos vinculados con el patrón de diseño MVVM, vamos a desarrollar una aplicación de tipo "libreta de direcciones". Este estudio le permitirá ver el uso de los bindings, la implementación de un `ViewModel` y, por último, la lógica que debe adoptar para mantener la separación entre la interfaz gráfica y la lógica de la aplicación. El almacenamiento de la información se realizará en un archivo de texto que utiliza la tabulación como separador entre dos datos de un registro. Tras el estudio de los capítulos dedicados

a ADO.NET y LINQ será capaz de crear su propia capa de persistencia para el almacenamiento de la información en una base de datos.

La aplicación que vamos a desarrollar tendrá el siguiente aspecto:



Estructura de la aplicación

La primera etapa del desarrollo se corresponde con la creación y la estructuración del proyecto.

- En primer lugar, cree un nuevo proyecto **aplicación WPF** llamado **LibretaDireccionesMVVM**.

La arquitectura MVVM permite, como convención, estructurar la aplicación creando tres carpetas que permiten aislar cada una de sus capas.

- Cree tres carpetas en el proyecto y llámelas **View**, **Model** y **ViewModel**.
- Mueva el archivo `MainWindow.xaml` a la carpeta `View` con una operación de arrastrar y colocar.

Para respetar las convenciones del desarrollo de C#, hay que modificar el espacio de nombres asociado a la clase `MainWindow`.

- En `MainWindow.xaml.cs`, remplace la siguiente línea:

```
namespace LibretaDireccionesMVVM
```

por:

```
namespace LibretaDireccionesMVVM.View
```

- En `MainWindow.xaml`, aplique la modificación del espacio de nombres modificando el nombre de la clase asociada al código XAML.

```
<Window x:Class="LibretaDireccionesMVVM.MainWindow"
```

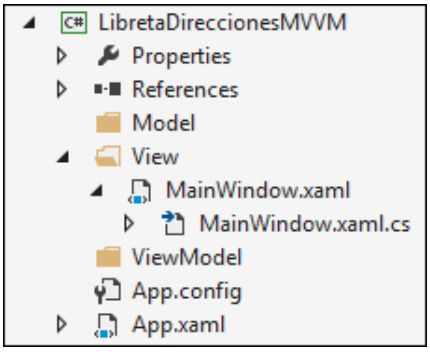
por:

```
<Window x:Class="LibretaDireccionesMVVM.View.MainWindow"
```

- En el archivo `App.xaml`, modifique el atributo `StartupUri` de la etiqueta `Application` de manera que refleje la nueva ruta de la página en el árbol de la aplicación.

```
StartupUri="View/MainWindow.xaml"
```

Llegados a este punto, la estructura de la aplicación en el **explorador de soluciones** es la siguiente:



La capa Model

Esta capa tiene como responsabilidad la estructuración y registro de los datos en una base de datos, en un archivo o, en el caso que nos ocupa, en memoria.

Conviene, en primer lugar, crear un tipo de datos que permita almacenar información relativa a un contacto: nombre, apellidos, dirección, etc.

- Agregue una clase llamada `ContactoModel` en la carpeta `Model` del proyecto.

```
internal class ContactoModel
{
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
    public string Teléfono { get; set; }
    public string Email { get; set; }
    public string Dirección { get; set; }
    public string Función { get; set; }
    public string Empresa { get; set;  }
}
```

Es necesario, también, para obtener una máxima separación de las responsabilidades, crear una clase cuyo objetivo sea almacenar y recuperar los distintos contactos registrados.

Agregue una clase llamada `GestorDeDatos` en la carpeta `Model` del proyecto.

Esta clase se encarga de leer y registrar los datos en el archivo *contactos.txt* que se encuentra en la misma carpeta que el archivo ejecutable.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

namespace LibretaDireccionesMVVM.Model
{
    internal class GestorDeDatos
    {
        private const string archivoDeDatos = "contactos.txt";

        public Contacto[] LeerTodosLosRegistros()
        {
            if (!File.Exists(archivoDeDatos))
                return new Contacto[0];

            string[] registros =
File.ReadAllLines(archivoDeDatos);

            Contacto[] resultado = new Contacto[registros.Length];

            for (int i = 0; i < registros.Length; i++)
            {
                string[] campos = registros[i].Split('\t');

                Contacto contacto = new Contacto
                {
                    Nombre = campos[0],
                    Apellidos = campos[1],
                    Función = campos[2],
                    Empresa = campos[3],
                    Teléfono = campos[4],
                    Email = campos[5],
                    Dirección1 = campos[6],
                    Dirección2 = campos[7],
                    CódigoPostal = campos[8],
                    Ciudad = campos[9],
                };
                resultado[i] = contacto;
            }

            return resultado;
        }

        public void Registrar(IEnumerable<Contacto> contactos)
        {
            StringBuilder builder = new StringBuilder();

            foreach (Contacto c in contactos)
```

```
            {
                string registro =
string.Format("{0}\t{1}\t{2}\t{3}\t{4}\t{5}\t{6}\t{7}\t{8}\t{9}",
c.Nombre, c.Apellidos, c.Función, c.Empresa, c.Teléfono, c.Email,
c.Dirección1, c.Dirección2, c.CódigoPostal, c.Ciudad);

                builder.AppendLine(registro);
            }


            File.WriteAllText(archivoDeDatos,  builder.ToString());
        }
    }
}
```

Pasemos a la creación de la interfaz gráfica.

- Modifique el archivo `MainWindow.xaml` de manera que disponga de una barra de herramientas con tres botones de acción, una `ListBox` y, por último, un `Grid` que contenga todos los campos editables.

```
<Window x:Class="LibretaDireccionesMVVM.View.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="450" Width="650" >

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="150" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <!-- Botones de acción (Agregar, Eliminar, Guardar Todo) -->
        <ToolBarTray Grid.Row="0" Grid.Column="0"
Grid.ColumnSpan="2">
            <ToolBar>
                <Button Content="Agregar" />
                <Button Content="Eliminar" />
                <Button Content="Guardar Todo" />
            </ToolBar>
        </ToolBarTray>

        <!-- Lista de contactos -->
        <ListBox x:Name="listBoxContactos" Grid.Row="1"
Grid.Column="0" />

        <!-- Detalle del contacto seleccionado -->
        <Grid Margin="0" Grid.Row="1" Grid.Column="1">
            <Grid.Resources>
                <Style TargetType="Label">
                    <Setter Property="Margin" Value="0,0,10,0" />
```

```
                    <Setter Property="VerticalAlignment"
Value="Center" />
                    <Setter Property="HorizontalAlignment"
Value="Right" />
                </Style>
                <Style TargetType="TextBox">
                    <Setter Property="Margin" Value="0,0,10,0" />
                    <Setter Property="VerticalAlignment"
Value="Center" />
                </Style>
            </Grid.Resources>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition  Height="Auto"/>
                <RowDefinition  Height="Auto"/>
                <RowDefinition  Height="Auto"/>
                <RowDefinition  Height="Auto"/>
                <RowDefinition  Height="Auto"/>
                <RowDefinition  Height="Auto"/>
                <RowDefinition  Height="Auto"/>
                <RowDefinition  Height="Auto" />
                <RowDefinition  Height="Auto" />
                <RowDefinition Height="Auto"/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Label Content="Nombre:" Grid.Row="1"  />
            <TextBox Grid.Column="1" Grid.Row="1"  />
            <Label Content="Apellidos:" Grid.Row="2" />
            <TextBox Grid.Column="1" Grid.Row="2"  />
            <Label  Content="Función:" Grid.Row="3"/>
            <TextBox Grid.Column="1" Grid.Row="3"  />
            <Label  Content="Empresa:" Grid.Row="4"/>
            <TextBox Grid.Column="1" Grid.Row="4"  />
            <Label Content="Teléfono:" Grid.Row="5" />
            <TextBox Grid.Column="1" Grid.Row="5" />
            <Label Content="Email:" Grid.Row="6" />
            <TextBox Grid.Column="1" Grid.Row="6" />
            <Label Content="Dirección 1:" Grid.Row="7" />
            <TextBox Grid.Column="1" Grid.Row="7" />
            <Label Content="Dirección 2:" Grid.Row="8" />
            <TextBox Grid.Column="1" Grid.Row="8" />
            <Label Content="Código Postal:" Grid.Row="9" />
            <TextBox Grid.Column="1" Grid.Row="9" />
            <Label Content="Ciudad:" Grid.Row="10"  />
            <TextBox Grid.Column="1" Grid.Row="10"  />
        </Grid>
    </Grid>
</Window>
```

Las etiquetas `Style` permiten agrupar ciertas propiedades relativas a un tipo de control. Es posible utilizar, en particular, estilos propios. También es posible aplicarlos automáticamente a todos los elementos correspondientes a su propiedad `TargetType`, siempre que estén situados más abajo en la rama del árbol. Esta última solución es la que se utiliza en esta ocasión. Un estilo permite especificar ciertas propiedades comunes a todos los objetos `Label`, mientras que otro estilo define los valores de las propiedades para los controles `TextBox`.

Llegados a este punto, vamos a trabajar simultáneamente sobre las capas `View` y `ViewModel` para mostrar los mecanismos utilizados para implementar una arquitectura MVVM.

- Cree una clase llamada `MainViewModel` en la carpeta `ViewModel` de la aplicación.

Esta clase llevará los datos a visualizar hasta la interfaz gráfica. Incluirá, también, otros elementos necesarios para gestionar las acciones que se ejecutan mediante los botones de la barra de herramientas, así como las propiedades que se utilizarán para otros procesamientos.

El primer punto a tener en cuenta es la recuperación de los datos almacenados mediante la clase `GestorDeDatos`. Esta operación se realiza en el constructor de la clase `MainViewModel` y alimenta una propiedad `ListaDeContactos`.

Esta propiedad es de tipo `ObservableCollection<Contacto>` pues, como hemos visto antes, esto permite realizar actualizaciones automáticas en la interfaz gráfica. Por el mismo motivo, vamos a implementar la interfaz `INotifyPropertyChanged` en esta clase.

```
public class MainViewModel : INotifyPropertyChanged
{
    private GestorDeDatos GestorDeDatos;

    private ObservableCollection<Contacto> listaContactos;

    public ObservableCollection<Contacto>  ListaContactos
    {
        get { return listaContactos; }
        set
        {
            listaContactos = value;
            OnPropertyChanged("ListaContactos");
        }
    }

    public MainViewModel()
    {
        GestorDeDatos = new GestorDeDatos();

        Contacto[] contactos =
GestorDeDatos.LeerTodosLosRegistros();
        ListaContactos = new
ObservableCollection<Contacto>(contactos);
    }

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
```

```
        {
            PropertyChanged(this, new
PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Para visualizar los contactos en la aplicación, es preciso definir un objeto `MainViewModel` como contexto de datos de la ventana.

- En el archivo `MainWindow.xaml.cs`, modifique el constructor de la clase de la siguiente manera:

```
public MainWindow()
{
   InitializeComponent();
   DataContext = new LibretaDireccionesMVVM.ViewModel.MainViewModel();
}
```

La línea que acabamos de agregar es la única que vinculará la parte `View` con la capa `ViewModel`.

Existen técnicas avanzadas que permiten no vincularlas directamente, pero no entran dentro del alcance de este libro.

Una vez asignado el valor del contexto de datos, implementaremos el `Binding` y el `DataTemplate` que permitirán visualizar los contactos en la `ListBox`.

- Modifique la definición del control `ListBox` como se describe a continuación.

```
<ListBox x:Name="listBoxContactos" Grid.Row="1" Grid.Column="0"
ItemsSource="{Binding ListaContactos}">
    <ListBox.ItemTemplate>
       <DataTemplate>
           <StackPanel Orientation="Vertical">
              <TextBlock Text="{Binding Nombre}" FontSize="20" />
               <TextBlock Text="{Binding Apellidos}" FontSize="20" />
              <TextBlock Text="{Binding Empresa}" FontSize="14" />
           </StackPanel>
       </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Agregar, eliminar y registrar los contactos se realiza desde la barra de herramientas. Sería posible crear un controlador para cada botón para el evento `Click`, pero la arquitectura MVVM recomienda utilizar **comandos**. Estos procesamientos, que no están vinculados a la interfaz gráfica, se encuentran en el `ViewModel`.

El control `Button` posee una propiedad `Command` a la que asignaremos valor mediante un `Binding`. Esta propiedad recibe como parámetro un valor de tipo `ICommand`, es decir, un objeto que implementa esta interfaz. El framework .NET no provee, desafortunadamente, ninguna implementación adecuada para nuestro ejemplo y tendremos que crear una en el proyecto. La implementación propuesta a continuación es muy sencilla, pero está perfectamente adaptada a nuestra aplicación.

Cree una clase `Command` en la carpeta `ViewModel` del proyecto.

```
using System;
using System.Windows.Input;

namespace LibretaDireccionesMVVM.ViewModel
{
    public class Command : Icommand
    {
        readonly Action<object> acciónAEjecutar;

        public Command(Action<object> execute)
            : this(execute, null)
        {
        }
        public Command(Action<object> execute, Predicate<object>
canExecute)
        {
            acciónAEjecutar = execute;
        }
        public bool CanExecute(object parameter)
        {
            return true;
        }

        public event EventHandler CanExecuteChanged;
        public void Execute(object parameter)
        {
            acciónAEjecutar(parameter);
        }
    }
}
```

Existen otras muchas implementaciones más o menos ricas, aparecidas con WPF. Una de las más utilizadas es la clase `RelayCommand` provista por el **MVVM Light Toolkit**, editado por Galasoft.

En la definición de la clase `MainViewModel`, agregue tres propiedades públicas de tipo `ICommand` (o `Command`) que se corresponderán con los tres botones de la barra de herramientas.

```
public ICommand ComandoNuevoContacto { get; set; }
public ICommand ComandoEliminarContacto { get; set; }
public ICommand ComandoGuardarTodo { get; set;  }
```

Cada uno de estos comandos requiere la creación de un método que contenga las instrucciones a ejecutar.

```
private void AcciónNuevoContacto(object parámetro)
{
   Contacto contacto = new Contacto { Nombre = "Nombre", Apellidos =
"Apellidos" };
    ListaContactos.Add(contacto);
}
```

```
private void AcciónGuardarTodo(object parámetro)
{
    GestorDeDatos.Registrar(ListaContactos);
}

private void AcciónEliminarContacto(object parámetro)
{

}
```

El procedimiento para eliminar un contacto no puede implementarse en el estado. En efecto, el objeto que se desea eliminar, que es el objeto seleccionado en la `ListBox`, no es conocido. La creación de una propiedad en el `ViewModel` y la inclusión de un `Binding` en la definición de la `ListBox` permiten resolver este problema.

```
private Contacto contactoSeleccionado;
public Contacto ContactoSeleccionado
{
    get { return contactoSeleccionado; }
    set
    {
          contactoSeleccionado = value;
          OnPropertyChanged("ContactoSeleccionado");
   }
}
```

```
<ListBox ... SelectedItem="{Binding ContactoSeleccionado,
Mode=TwoWay}">
```

Por último, podemos terminar la implementación del método `Acción EliminarContacto` y vincular los comandos a los botones mediante `Binding`.

```
private void AcciónEliminarContacto(object parámetro)
{
    if (ContactoSeleccionado != null)
           ListaContactos.Remove(ContactoSeleccionado);
}
```

```
<ToolBar>
    <Button Content="Agregar" Command="{Binding
ComandoNuevoContacto}" />
    <Button Content="Eliminar" Command="{Binding
ComandoEliminarContacto}" />
    <Button Content="Guardar Todo" Command="{Binding
ComandoGuardarTodo}" />
</ToolBar>
```

Llegados a este punto, es posible crear y eliminar registros, pero es imposible editarlos. Los campos de

información no están, en efecto, ligados a ningún contacto. Para remediar esta situación, hay que definir el contexto de estos elementos como el contacto seleccionado en la `ListBox`. Dotar de valor a la propiedad `DataContext` del `Grid` que contiene los campos de información parece una buena solución. El valor que se pasa es un `Binding` que apunta sobre la propiedad `SelectedItem` de la `ListBox`.

```
<Grid Margin="0" Grid.Row="1" Grid.Column="1"
      DataContext="{Binding  ElementName=listBoxContactos,
Path=SelectedItem}">
```

Puede, a continuación, asociar un `Binding` con cada control `TextBox` en modo `TwoWay` para asegurar que las modificaciones realizadas se reflejan en las propiedades correspondientes del objeto `Contacto`.

```
<Label Content="Nombre:" Grid.Row="1" />
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Nombre,
Mode=TwoWay}" />
<Label Content="Apellidos:" Grid.Row="2" />
<TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Apellidos,
Mode=TwoWay}" />
<Label Content="Función:" Grid.Row="3"/>
<TextBox Grid.Column="1" Grid.Row="3" Text="{Binding Función,
Mode=TwoWay}" />
<Label Content="Empresa:" Grid.Row="4"/>
<TextBox Grid.Column="1" Grid.Row="4" Text="{Binding Empresa,
Mode=TwoWay}" />
<Label Content="Teléfono:" Grid.Row="5"/>
<TextBox Grid.Column="1" Grid.Row="5" Text="{Binding Teléfono,
Mode=TwoWay}" />
<Label Content="Email:" Grid.Row="6"/>
<TextBox Grid.Column="1" Grid.Row="6" Text="{Binding Email,
Mode=TwoWay}" />
<Label Content="Dirección 1:" Grid.Row="7" />
<TextBox Grid.Column="1" Grid.Row="7" Text="{Binding Dirección1,
Mode=TwoWay}" />
<Label Content="Dirección 2:" Grid.Row="8" />
<TextBox Grid.Column="1" Grid.Row="8" Text="{Binding Dirección2,
Mode=TwoWay}" />
<Label Content="Código Postal:" Grid.Row="9" />
<TextBox Grid.Column="1" Grid.Row="9" Text="{Binding CódigoPostal,
Mode=TwoWay}" />
<Label Content="Ciudad:" Grid.Row="10" />
<TextBox Grid.Column="1" Grid.Row="10" Text="{Binding Ciudad,
Mode=TwoWay}" />
```

La aplicación es, ahora, funcional. Quedan por pulir algunos detalles:

- Es posible hacer clic sobre el botón **Eliminar** cuando no se ha seleccionado ningún contacto.
- Es posible, también, escribir un texto en los campos de información cuando no se ha seleccionado ningún contacto.

Una solución a este problema sería asignar valor a la propiedad `IsEnabled` de estos controles mediante un `Binding` que apunte sobre una propiedad booleana del `ViewModel`.

- Agregue la propiedad `ActivarEliminaciónYEdición` en el `ViewModel` y modifique la definición de la propiedad `Contacto Seleccionado` de manera que su modificación asigne valor a dicha propiedad `ActivarEliminaciónYEdición`.

```
private bool activarEliminaciónYEdición;

public bool ActivarEliminaciónYEdición
{
    get { return activarEliminaciónYEdición
    set
    {
          activarEliminaciónYEdición = value;
          OnPropertyChanged("ActivarEliminaciónYEdición");
    }
}

public Contacto ContactoSeleccionado
{
    get { return contactoSeleccionado; }
    set
    {
          contactoSeleccionado = value;
          OnPropertyChanged("ContactoSeleccionado");

          if (contactoSeleccionado == null)
                ActivarEliminaciónYEdición  = false;
          else
                ActivarEliminaciónYEdición = true;
    }
}
```

- Modifique la definición del botón de eliminación para tener en cuenta esta desactivación.

```
<Button Content="Eliminar" Command="{Binding
ComandoEliminarContacto}" IsEnabled="{Binding
ActivarEliminaciónYEdición}" />
```

Para los campos de información, la situación es ligeramente diferente: su contexto de datos no es el `ViewModel` sino un objeto `Contacto`. Para modificar localmente el contexto de datos es necesario utilizar un elemento cuyo contexto de datos sea el `ViewModel`. El objeto `Window` raíz es, de este modo, el objeto indicado para realizar esta operación.

Dé nombre al objeto `Window` mediante un atributo `x:Name` y, a continuación, cree el vínculo entre la propiedad `IsEnabled` y la propiedad `Activar-EliminaciónYEdición` en el estilo que se aplica a los controles `TextBox` para evitar tener que duplicar el código para cada `TextBox`.

```
<Window ... x:Name="ventana">
<Style TargetType="TextBox">
    <Setter Property="Margin" Value="0,0,10,0" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="IsEnabled" Value="{Binding
```

```
ElementName=ventana,
Path=DataContext.ActivarEliminaciónYEdición}"  />
</Style>
```

Acaba de realizar una aplicación de gestión de contactos de la A a la Z utilizando la arquitectura MVVM.

Si bien requiere un poco más de trabajo, así como un cambio en la manera de pensar, el uso de este patrón de diseño distribuye mucho mejor las distintas capas de la aplicación y facilita enormemente el mantenimiento del código.

El código fuente del ejemplo completo que se ha desarrollado en esta sección está disponible para su descarga desde la página Información.

Principios de una base de datos

Las bases de datos constituyen actualmente un elemento imprescindible para casi cualquier aplicación. Sustituyen al uso de archivos, pesados y complicados para el desarrollador. Las bases de datos permiten, también, compartir información de manera más sencilla, ofreciendo un excelente rendimiento.

El uso de bases de datos requiere el conocimiento de dos requisitos previos: ciertos elementos de la terminología que se utilizan con más frecuencia así como algunas nociones de manipulación de datos mediante el lenguaje SQL.

1. Terminología

El conocimiento de los términos descritos a continuación es imprescindible para la comprensión de este capítulo.

Tabla

Una tabla es una unidad lógica de almacenamiento que se corresponde con un tipo de dato. Su estructura se define mediante campos (columnas) y contiene registros (filas). El equivalente lógico de una tabla en C# es una clase.

Registro

Un registro es un elemento de una tabla que posee un valor para cada columna de la tabla. El equivalente en C# de un registro sería un objeto, es decir una instancia de una clase.

Campo

Un registro contiene varios campos. Cada uno de estos campos representa un dato compuesto que compone el registro. El equivalente en C# de un campo es una propiedad.

Estos tres elementos se representan, a menudo, bajo la forma de una tabla, lo que permite entender mejor y manipular estos conceptos.

Identificador	Nombre	Apellidos	FechaNacimiento
1	Enrique	MARTIN FERNANDEZ	1954-01-25 00:00:00:00.000
2	Eric	DIAZ CASADO	1982-05-04 00:00:00:00.000
3	Miguel	MARTIN ROJO	2004-05-28 00:00:00:00.000

Clave primaria

Una clave primaria es un **identificador único** para cada registro de una tabla. La base de datos la reconoce de modo que puede utilizarse en alguna otra tabla para hacer referencia a un registro.

Debe definirse de manera explícita, en general en el momento de creación de la tabla.

2. El lenguaje SQL

Los desarrolladores pueden manipular los datos almacenados en una base de datos relacional mediante el lenguaje SQL (*Structured Query Language*).

Este lenguaje se basa en cuatro tipos de instrucciones que se corresponden con las operaciones de lectura, agregación, modificación y eliminación de datos.

Define también ciertas instrucciones que permiten manipular la estructura de los datos: creación de tablas, manipulación de índices o gestión de las relaciones entre las tablas.

El objetivo de este libro no es realizar una descripción exhaustiva del lenguaje SQL, de modo que las instrucciones de manipulación de la estructura de los datos no se abordarán.

a. Búsqueda de registros

La lectura de datos se realiza mediante la palabra clave SELECT. Esta, asociada con la palabra clave WHERE, permite filtrar y recuperar los juegos de datos propios de una o varias tablas de la base de datos.

La consulta siguiente permite recuperar el conjunto de nombres y apellidos en la lista de registros de la tabla Cliente:

```
SELECT Nombre, Apellidos FROM Cliente
```

Es posible, también, utilizar el símbolo * como atajo sintáctico para designar el conjunto de campos de la tabla:

```
SELECT * FROM Cliente
```

En la mayoría de casos no es conveniente recuperar el conjunto completo de datos. Es necesario, en este caso, utilizar la palabra clave WHERE para restringir el número de registros recuperados en función de uno o varios predicados.

Cláusula WHERE

La cláusula WHERE permite especificar una o varias condiciones que deben respetar los registros para formar parte del resultado de la consulta SQL.

La siguiente consulta recupera la lista de clientes cuyo nombre es Antonio.

```
SELECT * FROM Cliente WHERE Nombre = 'Antonio'
```

Para filtrar a la vez sobre el nombre y los apellidos es necesario combinar dos condiciones mediante las palabras clave AND u OR.

```
SELECT * FROM Cliente WHERE Nombre = 'Antonio' AND Apellidos = 'Amaro  Sanz'
```

Es posible utilizar otros predicados en las cláusulas WHERE para verificar que un valor forma parte de un conjunto o comparar parcialmente cadenas de caracteres.

WHERE ... IN

El uso de la cláusula `WHERE ... IN` permite validar que un valor forma parte de un conjunto de valores específicos. Este conjunto puede estar predefinido o bien ser el resultado de otra consulta `SELECT`.

La lista de clientes habitantes de las ciudades de Madrid, Barcelona y Valencia puede recuperarse de la siguiente manera:

```
SELECT * FROM Cliente WHERE Ciudad IN ('Madrid', 'Barcelona', 'Valencia')
```

WHERE ... BETWEEN ... AND

Es posible, también, filtrar los registros comparando un valor con un conjunto especificado por sus extremos.

La búsqueda de clientes que habiten en la provincia de Madrid (28) podría realizarse mediante la siguiente consulta:

```
SELECT * FROM Cliente WHERE CódigoPostal BETWEEN 28000 AND 28999;
```

WHERE ... LIKE

En ciertos casos puede resultar interesante filtrar en base a un valor alfanumérico parcial, en particular para implementar escenarios de autocompletar. Para ello se utiliza la cláusula `WHERE ... LIKE`.

```
SELECT * FROM Cliente WHERE Nombre LIKE 'A%';
```

Aquí, la consulta busca la lista de clientes cuyo nombre empieza por una A mayúscula. El carácter % en este contexto simboliza un conjunto cualquiera de caracteres.

b. Agregar registros

Es posible agregar un registro a una base de datos mediante el comando `INSERT INTO`. Para ejecutarlo, es preciso especificar la tabla en la que se desean agregar los datos, enumerar los campos impactados y proporcionar los valores que se desean insertar en estos campos.

La siguiente línea agrega un registro en la tabla Cliente:

```
INSERT INTO Cliente (Nombre, Apellidos) VALUES ('Antonio', 'Amaro Sanz');
```

No especificar la lista de campos a completar equivale a enumerar la totalidad de los campos de la tabla. En este caso, es preciso proporcionar una lista de valores correspondientes al conjunto de estos campos. Es posible indicar el valor especial `NULL` para aquellos campos que no se desea informar, siempre y cuando la definición de dichos campos autorice este valor.

Considerando que la tabla Cliente contiene tres campos: Nombre, Apellidos y Dirección, podríamos escribir el ejemplo anterior de la siguiente manera:

```
INSERT INTO Cliente VALUES (’Antonio’, ’Amaro Sanz’, NULL);
```

c. Actualización de información

Es posible modificar información en la base de datos mediante una instrucción UPDATE. Este comando permite actualizar uno o varios campos de uno o varios registros.

Para ello, es preciso indicar el nombre de la tabla que debe actualizarse, seguido de la palabra clave SET y la lista de asignaciones de valores a ejecutar.

```
UPDATE Cliente SET Apellidos = ’JIMENEZ’;
```

Esta consulta modifica el campo Apellidos de todos los registros de la tabla Cliente, lo cual no es, por lo general, el comportamiento deseado. Para limitar el número de registros sobre los que realizar la actualización es posible utilizar una cláusula WHERE:

```
UPDATE Cliente SET Apellidos = ’JIMENEZ’ WHERE Apellidos = ’GIMENEZ’;
```

d. Eliminar información

Para eliminar uno o varios registros de una tabla es necesario utilizar el comando DELETE. Este comando se utiliza de una manera parecida a la instrucción SELECT.

Es posible eliminar los clientes cuyo nombre es Antonio mediante la siguiente instrucción:

```
DELETE FROM CLIENTE WHERE Nombre = ’Antonio’
```

ADO.NET

ADO.NET es un conjunto de tipos que proporciona el framework .NET cuyo objetivo común es manipular bases de datos. Este componente, incluido desde los inicios de .NET, es el fundamento del acceso a datos con C#.

1. Presentación

Los tipos que componen ADO.NET se dividen en dos categorías, definidas de manera que resulta muy sencillo separar las consultas a la base de datos de la manipulación de los datos en la aplicación.

Los procesamientos que deben realizarse directamente sobre una base de datos se implementan mediante tipos específicos de un proveedor de datos. Es posible, utilizando estos tipos, recuperar juegos de datos o eliminar registros, por ejemplo.

Los juegos de datos recuperados pueden, también, almacenarse de manera local y es posible manipularlos de manera completamente independiente de su fuente, gracias a ciertos tipos dedicados. Este modo de funcionamiento se denomina **modo desconectado**. Permite, en particular, producir secciones de código comunes a todos los tipos de bases de datos, lo que mejora la fiabilidad y la mantenibilidad de las aplicaciones reduciendo el número de líneas de código y, en consecuencia, el número de potenciales fuentes de error.

2. Los proveedores de datos

Los proveedores de datos son elementos que permiten establecer un diálogo con los distintos tipos de bases de datos. Cada uno de estos proveedores se diseña para autorizar el funcionamiento con un tipo de base de datos.

Existen cuatro proveedores que se incluyen con el framework .NET. Estos permiten interactuar con las fuentes de datos siguientes:

- SQL Server
- Oracle
- OLE DB
- ODBC

Las funcionalidades que presenta cada uno de estos proveedores son comparables. En efecto, las clases que contienen heredan de tipos comunes que permiten, en particular, establecer conexiones, ejecutar consultas o leer datos. Los principales tipos básicos implementados por estos proveedores son los siguientes:

- La clase `Connection` permite establecer una conexión a una base de datos.
- La clase `DbCommand` permite ejecutar una o varias consultas SQL.
- La clase `DbParameter` representa un parámetro de un comando.
- La clase `DbDataReader` proporciona una acceso secuencial (hacia adelante, únicamente) y de solo lectura a los juegos de datos.
- La clase `DbDataAdapter` es la pasarela entre los modos conectado y desconectado. Asegura el almacenamiento en la caché local de los datos que provienen de la ejecución de consultas SQL y actualiza los datos de la base de datos en función del estado de los datos locales de trabajo.
- La clase `DbTransaction` encapsula la gestión de las transacciones para ejecutar consultas SQL múltiples.

Este modo de estructuración ofrece la posibilidad de extender los orígenes de datos soportados por una aplicación

desarrollando proveedores propios para otros sistemas de bases de datos. Hoy en día existen proveedores para MySQL, SQLite, PostgreSQL, Firebird y muchos otros sistemas de bases de datos.

a. SQL Server

El proveedor de datos para SQL Server utiliza su propio protocolo para comunicarse con el servidor de bases de datos. Es, por ello, **poco exigente** en recursos y está **optimizado**, pues no utiliza ninguna capa de software suplementaria como hacen OLE DB y ODBC. Es compatible con las versiones 7.0 (aparecida en 1998) y superiores de SQL Server.

Las clases específicas de este proveedor se encuentran en el espacio de nombres `System.Data.SqlClient`. Su nombre está, sistemáticamente, prefijado por `Sql`: `SqlConnection`, `SqlDataReader`, etc.

Para las explicaciones y ejemplos de este libro se ha utilizado este proveedor de datos, pero sería posible trasladarlos a otros proveedores utilizando las clases adecuadas.

b. Oracle

El proveedor de datos para Oracle permite establecer la comunicación con bases de datos Oracle 8.1.7 y superiores a través de la capa de software cliente de Oracle. Es necesario, por tanto, instalar los **componentes Oracle** apropiados para poder utilizar este proveedor.

Los tipos proporcionados por este proveedor se encuentran en el espacio de nombres `System.Data.OracleClient`, declarado en el ensamblado System.Data.OracleClient.dll, lo cual implica agregar una referencia a dicho ensamblado para poder utilizar este proveedor. El nombre de cada uno de estos tipos está prefijado por Oracle.

El proveedor Oracle se provee con el framework .NET desde la versión 2.0 hasta la versión 4.7. De todos modos, se considera como **depreciado** desde la versión 4.0, lo que significa que el framework aún lo soporta, pero que podría eliminarse y desaparecer en la siguiente versión. Microsoft recomienda utilizar proveedores de datos de terceros para comunicarse con bases de datos Oracle.

c. OLE DB

Este proveedor utiliza la capa de software OLE DB para comunicarse con los servidores de bases de datos. Resulta interesante cuando algún proveedor específico a una base de datos no existe, pero sí se dispone de un controlador nativo compatible con OLE DB. En este caso, el proveedor OLE DB va a **interactuar como un intermediario entre la aplicación y el controlador**, el cual se ocupará de realizar la comunicación con la base de datos.

Las clases específicas a este proveedor de bases de datos se encuentran en el espacio de nombres `System.Data.OleDb` y su nombre está prefijado por `OleDb`.

Para poder funcionar, este proveedor requiere la presencia en la máquina de los componentes MDAC (*Microsoft Data Access Components*) en versión 2.6 o superior.

d. ODBC

El proveedor ODBC utiliza el mismo principio de funcionamiento que el proveedor OLE DB. Requiere la existencia de un controlador nativo compatible con ODBC para el origen de datos correspondiente para poder dialogar con él. A continuación, el controlador nativo se comunicará con la base de datos.

Este proveedor de datos está implementado en el espacio de nombres `System.Data.Odbc` y cada uno de sus tipos tiene como prefijo `Odbc`.

Para poder funcionar, este proveedor requiere la presencia en la máquina de los componentes MDAC (*Microsoft Data Access Components*) en versión 2.6 o superior.

Utilizar ADO.NET en modo conectado

El modo conectado es el modo de funcionamiento original de las bases de datos. Permite mantener la conexión a la base de datos permanentemente, lo que presenta las siguientes ventajas:

- Su uso es muy sencillo. En efecto, en este modo la conexión al origen de datos se crea tras el inicio de la aplicación y se destruye cuando el usuario sale.
- Permite disponer permanentemente de datos actualizados.
- Permite gestionar de manera más sencilla los accesos secuenciales. Dado que los usuarios están permanentemente conectados resulta más sencillo determinar quién utiliza los datos.

El modo conectado presenta, no obstante, algunos problemas nada despreciables:

- Impone que exista una conexión de red permanente entre el cliente y el servidor de bases de datos. A la hora de la movilidad, resulta especialmente delicado imponer esta restricción a un usuario.
- El servidor de base de datos debe gestionar múltiples conexiones permanentes simultáneamente, lo cual puede provocar un uso excesivo de recursos.

1. Conexión a una base de datos

Antes de ejecutar cualquier consulta es necesario abrir una conexión con el servidor de bases de datos. Esta conexión se crea mediante la clase `SqlConnection`. Tras la instanciación de una conexión es necesario inicializar ciertos valores relativos a la base de datos. Estos se especifican mediante la cadena de conexión.

a. Cadenas de conexión

Una cadena de conexión es una cadena de caracteres que contiene la información necesaria para establecer una conexión con una base de datos. Esta información se especifica en forma de parejas de campos clave/valor separados por puntos y coma, las claves son nombres de parámetro reconocidos por el proveedor de datos.

Cuando se asigna a la propiedad `ConnectionString` de un objeto `SqlConnection`, se realiza un análisis de la cadena de conexión. Tras este análisis, se extraen los distintos valores contenidos en ella y se asignan a las distintas propiedades de la conexión. En caso de error durante esta etapa de análisis, se produce una excepción y las propiedades de la conexión permanecen sin cambios.

Las siguientes palabras clave están disponibles para configurar una cadena de conexión:

Data Source

Nombre o dirección de red del servidor al que se desea establecer la conexión. Es posible especificar un número de puerto a continuación, separando los valores mediante una coma. El valor por defecto del número de puerto es el 1.433.

```
Data Source=SRVSQL
```

Initial Catalog

Nombre de la base de datos sobre la que se desea realizar la conexión.

```
Initial Catalog=Northwind
```

User Id

Nombre del usuario de SQL Server que se desea utilizar para establecer la conexión. Cualquier consulta ejecutada a través de la conexión se someterá a los permisos de acceso correspondientes a esta cuenta de usuario.

```
User Id=sqluser
```

Password

Contraseña asociada a la cuenta de usuario de SQL Server utilizada.

```
Password=usuario7
```

Integrated Security

Valor booleano que ofrece la posibilidad de utilizar la autenticación de Windows en curso para la conexión.

```
Integrated Security=True
```

Connect Timeout

Tiempo (en segundos) durante el cual la aplicación espera a que se establezca la conexión. En caso de superarse este intervalo de tiempo, se produce una excepción. Por defecto, este intervalo es de 15 segundos.

```
Connect Timeout=30
```

Min Pool Size

Número mínimo de conexiones que se desea conservar en el pool de conexiones.

```
Min Pool Size=20
```

Max Pool Size

Número máximo de conexiones que se desea conservar en el pool de conexiones.

```
Max Pool Size=70
```

La noción de pool de conexiones se aborda con detalle un poco más adelante en este mismo capítulo.

De este modo, para establecer una conexión al servidor **SRVSQL**, para la base de datos **Northwind** con el nombre de usuario **sqluser** y la contraseña **usuario7**, utilizaremos la siguiente cadena de conexión:

```
string cadena = "Data Source=SRVSQL;Initial
Catalog=Northwind;User  Id=sqluser;Pwd=usuario7";
```

Puede resultar delicado dominar la totalidad de parámetros que pueden utilizarse en una cadena de conexión. La clase `SqlConnectionStringBuilder` tiene como objetivo simplificar la creación de una cadena de conexión. Este tipo posee varias propiedades que se corresponden, cada una, con un parámetro que puede utilizarse en una cadena de conexión. Estas propiedades están documentadas de cara a comprender fácilmente su objetivo. De este modo, no es necesario conocer a la perfección cada palabra clave utilizada para obtener una cadena de conexión funcional.

```
SqlConnectionStringBuilder builder = new
SqlConnectionStringBuilder();
builder.DataSource = "SRVSQL";
builder.InitialCatalog = "Northwind";
builder.UserID = "sqluser";
builder.Password = "usuario7";

Console.WriteLine("Cadena de conexión: {0}",
builder.ConnectionString);
```

El resultado de la ejecución de este código es el siguiente:

```
Cadena de conexión: Data Source=SRVSQL;Initial
Catalog=Northwind;User  ID=sqluser;Password=usuario7
```

b. Pool de conexiones

Los pools de conexiones permiten mejorar el rendimiento de una aplicación evitando la creación de conexiones, siempre que sea posible.

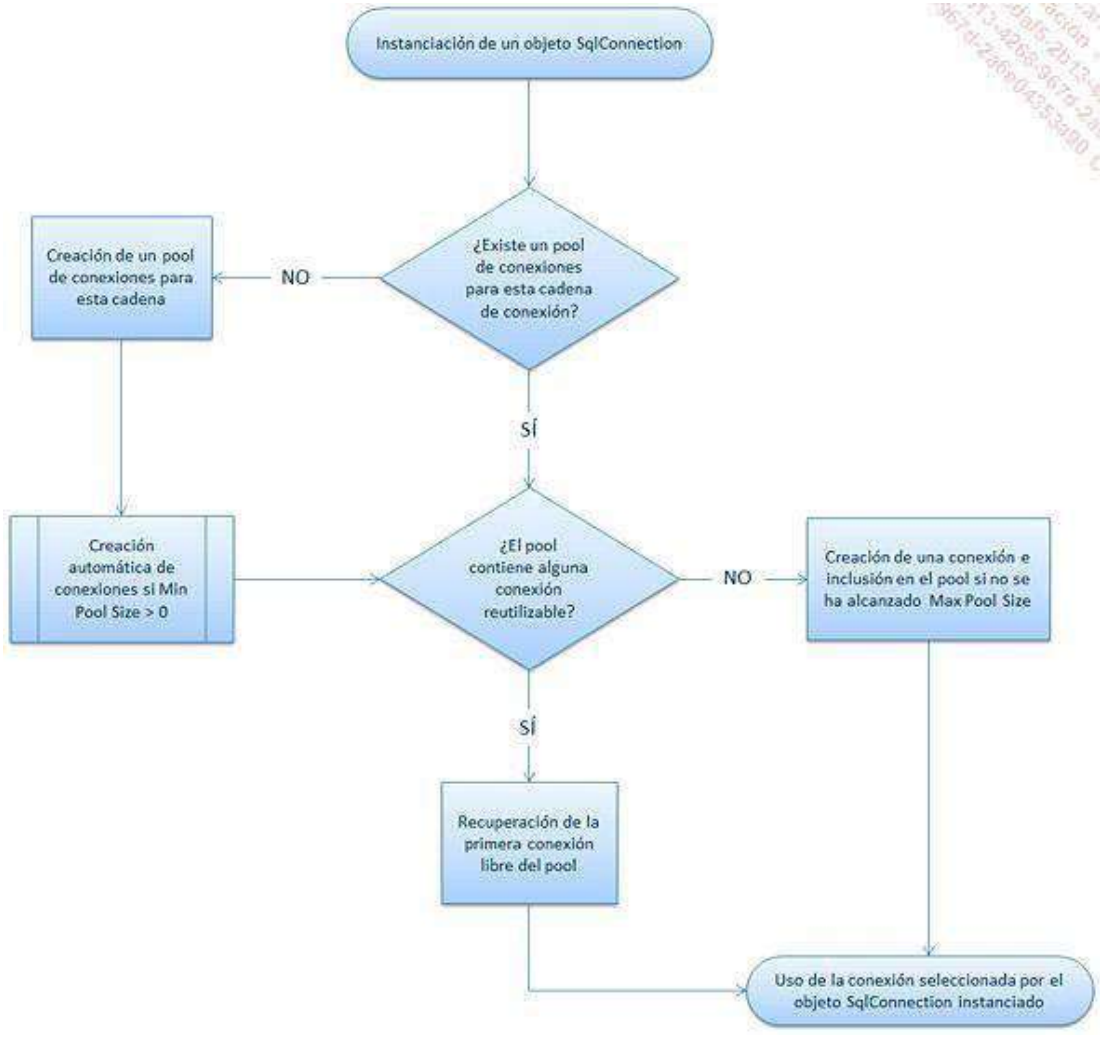
Establecer una conexión supone una acción costosa en términos de recursos, tanto del lado cliente como del lado del servidor. El principio del pool de conexiones es la implementación en caché de conexiones para su máxima reutilización. Según este objetivo, un pool de conexiones permite conservar una cierta cantidad de conexiones abiertas que utilizan la misma cadena de conexión. Cuando la aplicación inicia una conexión a una base de datos, la primera conexión disponible en el pool asociado a la cadena de conexión especificada se utiliza automáticamente.

Los pools creados de esta forma existen mientras la aplicación se esté ejecutando.

Para cada uno de estos pools es posible especificar, mediante las propiedades `Min Pool Size` y `Max Pool Size` de la cadena de conexión, los números mínimo y máximo de conexiones que debe contener.

Si se alcanza el número máximo de conexiones indicado, y no existe ninguna conexión disponible en el pool, la solicitud se quedará en espera hasta que haya disponible alguna conexión. Una conexión se pone a disposición del pool cuando se cierra explícitamente en el código mediante el método `Close`, o bien cuando se invoca al método `Dispose` de la conexión. Para asegurar el correcto reciclaje de las conexiones es importante, por tanto, cerrar las conexiones que ya no se estén utilizando.

A continuación se muestra un esquema del uso de un pool de conexiones.



c. Gestión de la conexión

Establecimiento de la conexión

Para crear una conexión a una base de datos es necesario instanciar un objeto `SqlConnection` e inicializar su cadena de conexión, lo cual puede realizarse de dos maneras:

- Mediante el parámetro `connectionString` del constructor de la clase `SqlConnection`.
- Asignando un valor a la propiedad `ConnectionString` del objeto `SqlConnection`. Esta asignación solo puede realizarse cuando la conexión está cerrada.

```
string cadenaDeConexión = "Data Source=SRVSQL;Initial
Catalog=Northwind;User  ID=sqluser;Password=usuario7";
```

```
SqlConnection conexión;

//Inicialización mediante el constructor
conexión = new  SqlConnection(cadenaDeConexión);

//O inicialización mediante la propiedad
conexión = new SqlConnection();
conexión ConnectionString = cadenaDeConexión;
```

Una vez definido el destino de la conexión es necesario invocar al método `Open` del objeto `conexión` para abrir la conexión entre la aplicación y el servidor de bases de datos.

```
conexión.Open();
```

Cierre de la conexión

El cierre de una conexión es una etapa muy importante. En efecto, una conexión que no se cierra es una conexión potencialmente perdida que debe, no obstante, gestionarse por parte del servidor de bases de datos.

Este cierre debe realizarse mediante el método `Close`:

```
conexión.Close();
```

La clase `SqlConnection` implementa la interfaz `IDisposable`. Es conveniente, por tanto, utilizar la estructura `using` (consulte el capítulo Las bases del lenguaje, sección Otras estructuras) para evitar cualquier bloqueo de la conexión en caso de producirse alguna excepción u olvidar realizar su cierre.

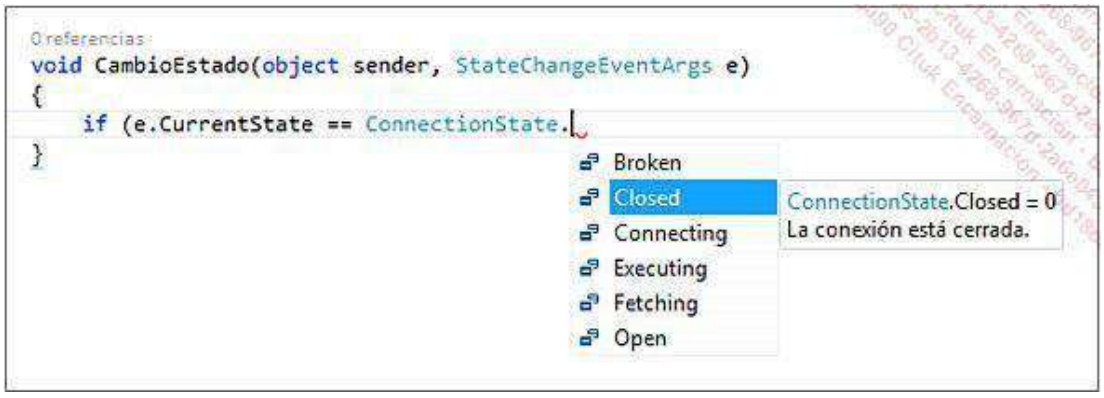
Eventos

La clase `SqlConnection` posee dos eventos que permiten obtener información relativa a la conexión.

El evento `StateChanged` se produce cuando se modifica el estado de la conexión, modificación realizada por parte del usuario o forzada por parte del servidor de bases de datos. El controlador para estos eventos tiene la siguiente firma:

```
void CambioEstado(object sender, StateChangeEventArgs e)
```

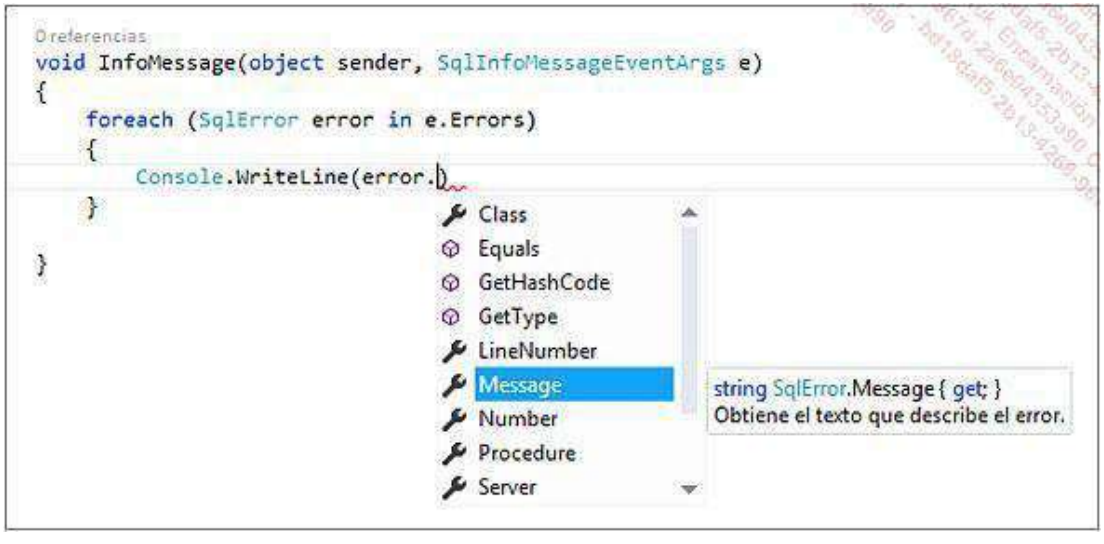
El parámetro de tipo `StateChangeEventArgs` tiene dos propiedades: `OriginalState` y `CurrentState`. Estas permiten conocer, respectivamente, el estado original de la conexión y su nuevo estado. Estas propiedades contienen valores de la enumeración `ConnectionState` y es posible comprobar fácilmente el valor de estos elementos para actuar en consecuencia.



El evento `InfoMessage` permite obtener información relativa a cualquier situación anormal pero que no sea crítica (con una severidad inferior a 10) que se produzca sobre la conexión. La firma del controlador de eventos asociado es la siguiente:

```
void InfoMessage(object sender, SqlInfoMessageEventArgs e)
```

El parámetro de tipo `SqlInfoMessageEventArgs` contiene una colección de objetos `SqlErrors` que se corresponden, cada uno, con los errores devueltos por el servidor de bases de datos. Estos objetos tienen varias propiedades que permiten diagnosticar y corregir el problema.



2. Creación y ejecución de comandos

Una vez establecida la conexión al servidor de bases de datos, es posible ejecutar comandos SQL mediante objetos de tipo `SqlCommand`.

Estos comandos pueden corresponderse con cualquier tipo de consulta SQL (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE`, `ALTER TABLE`, etc.) pero, también, con la solicitud de ejecución de un procedimiento almacenado.

a. Definición y creación de un comando

La creación de un objeto `SqlCommand` se realiza de dos maneras:

- Utilizando uno de los constructores de la clase `SqlCommand`. Aquí utilizaremos el constructor sin parámetro, aunque es posible, evidentemente, utilizar cualquiera de los otros tres, siempre y cuando se definan los parámetros necesarios para su uso.

```
SqlCommand comando = new SqlCommand();
```

Para asociar este comando a una conexión se asigna un objeto `SqlConnection` a la propiedad `Connection` del comando.

```
comando.Connection = conexión;
```

- Utilizando el método `CreateCommand` del objeto `SqlConnection`.

```
SqlCommand comando = conexión.CreateCommand();
```

Preste atención: para poder ejecutar el comando es necesario que la conexión esté abierta. Compruebe su estado antes de crear los comandos que necesite, esto le evitará inconvenientes.

b. Selección de datos

La selección de datos es la operación que se realiza con mayor frecuencia sobre una base de datos. Esta es muy sencilla de implementar mediante los objetos `SqlConnection` y `SqlCommand`.

Tras la creación y la asociación de un objeto `SqlCommand` a una conexión, basta con asignar valor a la propiedad `CommandText` del comando con una consulta SQL para tener un comando de selección listo para ejecutarse.

```
comando.CommandText = "SELECT * FROM Alumnos";
```

La ejecución de consultas de selección de datos se realiza gracias al método `ExecuteReader` del objeto `SqlCommand`. Este método devuelve un objeto de tipo `SqlDataReader` gracias al cual es posible recuperar cada uno de los registros del resultado de la consulta, uno por uno, y únicamente avanzando en la colección de registros.

```
SqlDataReader reader = comando.ExecuteReader();
```

La lectura de un registro se realiza mediante la llamada al método `Read` del objeto `SqlDataReader`. Este método devuelve un valor booleano que indica si ha sido posible leer algún registro. Si es así, el objeto `SqlDataReader` contiene los datos del registro, que pueden manipularse mediante el operador `[]`.

```
reader.Read();
//Lectura del valor de la columna "Nombre" del registro
string nombre = (string)reader["Nombre"];
```

Es necesario realizar una conversión de tipo de los valores recuperados de esta manera, pues están expuestos

mediante el objeto SqlDataReader con forma de objeto object.

Es posible, a su vez, recuperar el valor deseado fuertemente tipado mediante alguno de los métodos especializados de la clase SqlDataReader: el método GetString para obtener una cadena de caracteres, GetInt32 para obtener un valor entero, GetDateTime para obtener una fecha, etc. Estos métodos reciben como parámetro un valor entero que representa el índice de la columna en el registro.

```
reader.Read();
//Lectura del valor de la columna situada en la primera posición
//del registro
string nombre = reader.GetString(0)
```

La lectura de la totalidad de los datos devueltos se realiza mediante un bucle while donde la condición de entrada es el valor devuelto por la llamada al método Read.

```
while (reader.Read())
{
    Console.WriteLine("Nombre = {0}", reader.GetString(0));
}
```

Cuando la consulta no devuelve ningún valor, la lectura del resultado puede simplificarse utilizando el método ExecuteScalar del comando. Este devuelve un valor de tipo object que debe convertirse al tipo deseado.

```
SqlCommand comando = conexión.CreateCommand();
comando.CommandText = "SELECT COUNT(*) FROM  Alumnos";

int numAlumnos = (int)comando.ExecuteScalar();
Console.WriteLine("Hay {0} alumnos", numAlumnos);
```

c. Acciones sobre los datos

La ejecución de una consulta para agregar, eliminar o actualizar datos se realiza de forma similar a la selección de datos. Se crea un comando que contiene una consulta UPDATE, INSERT o DELETE y se asocia a una conexión.

```
SqlCommand comando = conexión.CreateCommand();
comando.CommandText = "INSERT INTO Alumnos (Nombre, Apellidos) VALUES
('Antonio', 'AMARO SANZ')";
```

Una vez definido el comando, la llamada al método ExecuteNonQuery del objeto SqlCommand envía la consulta al servidor de bases de datos. El valor devuelto es un valor entero que representa el número de registros afectados por la consulta.

```
int numeroRegistrosAfectados = comando.ExecuteNonQuery();
```

Los comandos que modifican elementos de la estructura de la base de datos (ALTER TABLE, por ejemplo) se ejecutan, también, mediante este método.

d. Parametrización de un comando

La manera más sencilla de construir consultas SQL es la **concatenación de cadenas de caracteres**. Este método presenta la ventaja de que se comprende rápidamente, aunque resulta delicado de implementar. En efecto, provoca a menudo **errores de sintaxis** (se olvida algún espacio o alguna coma, por ejemplo) y puede presentar **problemas de seguridad** (inyección SQL).

El uso de **parámetros** permite reducir el número de problemas generados por la creación de una consulta SQL:

- Una consulta parametrizada puede **definirse sin concatenación**, lo cual reduce el riesgo de cometer algún error de sintaxis.
- **Los valores de los parámetros se validan** en cuanto a su tipo y a su longitud antes de ejecutar la consulta. Además, las cadenas de caracteres se escapan, de manera que no es posible interpretarlas como instrucciones SQL.

Estos dos puntos reducen considerablemente la superficie de ataque derivada de las inyecciones de SQL.

A continuación se muestra un ejemplo de construcción de un comando SQL a partir de la concatenación de cadenas:

```
string nombre = "Antonio";
string apellidos = "AMARO  SANZ";
DateTime fechanacimiento = new DateTime(2004, 05, 28);

SqlCommand comando = conexión.CreateCommand();
comando.CommandText = "INSERT INTO Alumnos (Nombre, Apellidos,
FechaNacimiento) VALUES (’" + nombre + "’, ’" + apellidos + "’, ’" +
fechanacimiento.ToString("dd/MM/yyyy") + "’)";
```

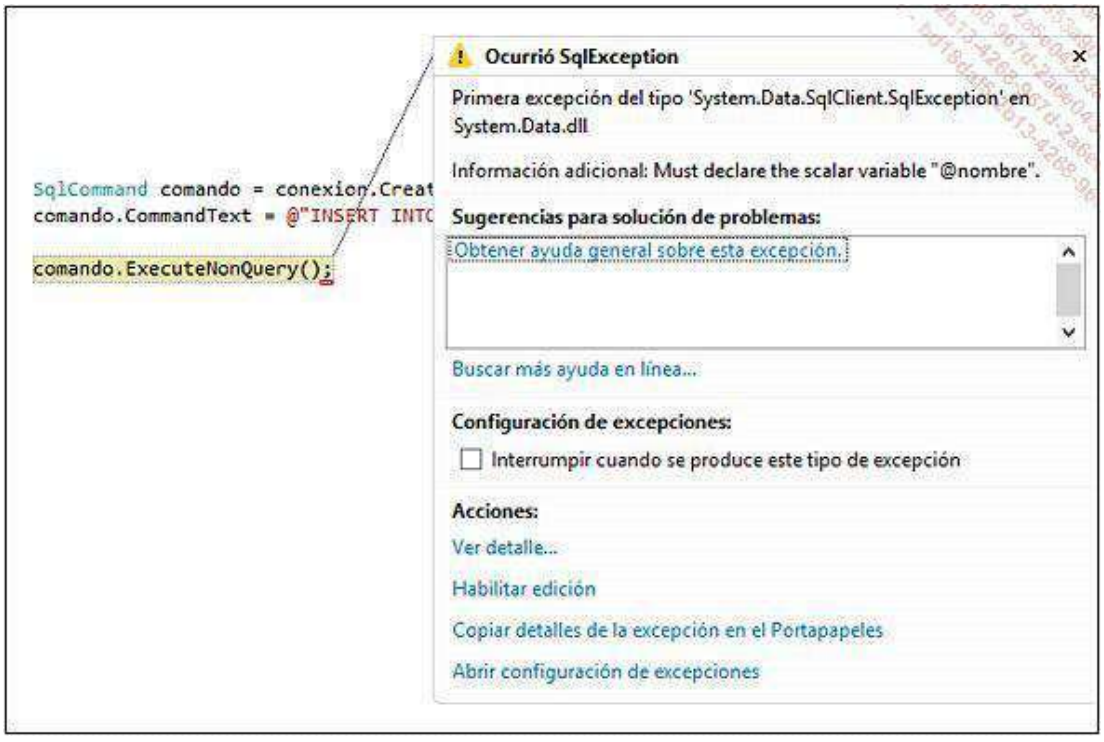
Las concatenaciones hacen que la lectura sea algo difícil y en función de la cultura utilizada por el servidor de bases de datos es posible obtener un comportamiento inesperado (o incluso una excepción) derivado del formato de la fecha.

Es posible reescribir esta construcción del comando utilizando parámetros de manera que resulte más fácil de mantener y más seguro.

En primer lugar, escribiremos la consulta definiendo el nombre y la ubicación de sus parámetros: `@nombre`, `@apellidos`, `@fechanacimiento`.

```
SqlCommand comando =  conexión.CreateCommand();
comando.CommandText = "INSERT INTO Alumnos (Nombre, Apellidos,
FechaNacimiento) VALUES (@nombre, @apellidos, @fechanacimiento)";
```

Si ejecutáramos el comando en este punto, se produciría una excepción, pues los valores de los parámetros no se han especificado.



La creación de parámetros se corresponde con la instanciación de objetos de tipo `SqlParameter` y su configuración. Es posible, en particular, especificar el tipo esperado en la base de datos mediante la propiedad `DbType` o el tamaño máximo del parámetro cuando se trata de una cadena de caracteres.

Es posible definir el parámetro como de solo lectura (`ParameterDirection.Input`), de solo escritura (`ParameterDirection.Ouput`) o de lectura y escritura (`ParameterDirection.InputOutput`) mediante la propiedad `Direction`. Esto puede resultar bastante útil en el caso de la ejecución de una función o procedimiento almacenado.

```
string nombre = "Antonio";
string apellidos = "AMARO  SANZ";
DateTime fechanacimiento = new DateTime(2004, 05, 28);

//Creación del parámetro @nombre
SqlParameter paramNombre = new SqlParameter("@nombre", nombre);
paramNombre.Direction = ParameterDirection.Input;
paramNombre.DbType = DbType.String;
paramNombre.Size = 30;

//Creación del parámetro @apellidos
SqlParameter paramApellidos = new SqlParameter("@apellidos",
apellidos);
paramApellidos.Direction = ParameterDirection.Input;
paramApellidos.DbType = DbType.String;
paramApellidos.Size = 30;

//Creación del parámetro @fechanacimiento
SqlParameter paramFecha = new SqlParameter("@fechanacimiento",
fechanacimiento);
paramFecha.Direction = ParameterDirection.Input;
paramFecha.DbType = DbType.DateTime;
```

Para terminar, conviene agregar los parámetros creados a la colección de parámetros del comando.

```
comando.Parameters.Add(paramNombre);
comando.Parameters.Add(paramApellidos);
comando.Parameters.Add(paramFecha);
```

e. Ejecución de procedimientos almacenados

Los procedimientos almacenados son elementos comparables a los procedimientos en C#, pero **se ejecutan completamente en el servidor de bases de datos**. Están formados por instrucciones SQL (o su variante específica del servidor de bases de datos) y pueden recibir parámetros, devolver valores y realizar procesamientos complejos.

Su uso presenta varias ventajas, en particular la **centralización del código SQL**, lo que permite simplificar el mantenimiento cuando evoluciona una base de datos sin impactar al código de las aplicaciones que la utilizan. Además, los procedimientos almacenados aprovechan un procesamiento particular: se **compilan** tras su primera ejecución en el motor de bases de datos y se alojan en caché, lo cual permite aportar una mejora de rendimiento en el caso de procesamientos complejos.

La llamada a un procedimiento almacenado desde C# es parecida a la ejecución de consultas SQL. Tras crear un objeto `SqlCommand`, se asigna a su propiedad `CommandText` el nombre del procedimiento almacenado que se desea ejecutar y se modifica la propiedad `CommandType` con el valor `CommandType.StoredProcedure` para indicar que `CommandText` contiene el nombre de un procedimiento almacenado.

A continuación, se especifican los parámetros esperados por el procedimiento almacenado de forma similar a como se hace con una consulta SQL. Cuando un procedimiento almacenado devuelve un valor (mediante la instrucción `RETURN`), conviene agregar un parámetro suplementario llamado `RETURN_VALUE` asignando a su propiedad `Direction` el valor `ParameterDirection.ReturnValue`. Tras ejecutar el procedimiento almacenado será posible, de este modo, recuperar el valor de retorno, como con las propiedades de escritura, a través de la propiedad `Value` del parámetro.

Para probar estas nociones vamos a utilizar un procedimiento almacenado que calcula la edad media de los alumnos nacidos a partir de un año concreto.

```
CREATE PROCEDURE CalculaEdadMediaAlumnosNacidosDespuesDelAño @año int AS
   -- Declaración de la variable que contendrá la edad media
   DECLARE @media decimal

   -- Cálculo de la  media
   SELECT @media = AVG((Year(GetDate()) -
Year(FechaNacimiento)))
          FROM Alumnos
          WHERE YEAR(FechaNacimiento) > @año

    --Se devuelve la variable @media
    RETURN @media
```

El código C# que permite invocar a este procedimiento almacenado es el siguiente.

```
int año = 2000;

SqlCommand comando = conexión.CreateCommand();
comando.CommandText = "CalculaEdadMediaAlumnosNacidosDespuesDelAño";
comando.CommandType = CommandType.StoredProcedure;

SqlParameter parámetroAño = new SqlParameter("@año", año);
parámetroAño.Direction = ParameterDirection.Input;
parámetroAño.DbType = DbType.Int32;

SqlParameter parámetroValorRetorno = new
SqlParameter("RETURN_VALUE", SqlDbType.Decimal);
parámetroValorRetorno.Direction = ParameterDirection.ReturnValue;

comando.Parameters.Add(parámetroAño);
comando.Parameters.Add(parámetroValorRetorno);

comando.ExecuteNonQuery();

Console.WriteLine("La edad media de los alumnos nacidos
después de {0} es {1} años", año, parámetroValorRetorno.Value);
```

Utilizar ADO.NET en modo desconectado

En el modo desconectado, la conexión al servidor de bases de datos no es permanente. Esto permite liberar recursos que podrían utilizarse por otra aplicación o por otro cliente de la base de datos.

Este modo de funcionamiento implica que hay que conservar una copia local de los datos sobre los que se desea trabajar. Para ello, es posible recrear de manera local una estructura parecida a la de una base de datos.

1. DataSet y DataTable

Las clases principales que permiten trabajar en modo desconectado son `DataSet`, `DataTable`, `DataRow` y `DataColumn`.

a. Descripción

Las distintas clases utilizadas para el modo desconectado son las contrapartes .NET de los elementos de la estructura de una base de datos.

DataSet

Un objeto `DataSet` es un contenedor, similar lógicamente a una base de datos. Es el que contendrá la totalidad de datos del conjunto sobre el que se desea trabajar.

DataTable

El tipo `DataTable` representa, como su propio nombre indica, una tabla. Un objeto `DataTable`, si bien puede existir independientemente, está lógicamente contenido en un `DataSet`.

DataColumn

La clase `DataColumn` es el equivalente .NET de las columnas SQL. Los `DataTable` poseen varios `DataColumn`, como las tablas en base de datos poseen varias columnas.

DataRow

Los objetos `DataRow` son los verdaderos contenedores de los datos. Cada `DataRow` se corresponde con un registro en un `DataTable`.

Estas cuatro clases no son los únicos tipos que representan elementos de la estructura de una base de datos. Encontramos, también, los tipos `UniqueConstraint`, `ForeignKeyConstraint` o `DataRelation`, que representan respectivamente las restricciones de unicidad, las claves foráneas y las relaciones entre objetos `DataTable`.

b. Llenar un DataSet a partir de una base de datos

Para trabajar con los datos de manera local es necesario recuperar los datos del servidor en un `DataSet`. Para ello, cada proveedor de datos proporciona un `DbDataAdapter` que permite realizar intercambios bidireccionalmente entre la base de datos y el `DataSet`. La implementación de la clase `DbDataAdapter` del proveedor de datos SQL Server es `SqlDataAdapter`.

Uso de un DbDataAdapter

En primer lugar, es preciso instanciar un objeto de un tipo derivado de la clase `DbDataAdapter`.

```
SqlDataAdapter adaptador = new SqlDataAdapter();
```

Este objeto posee una propiedad `SelectCommand` a la que se le asigna un comando de selección de datos. Es gracias a los datos devueltos por este comando como el objeto `SqlDataAdapter` podrá llenar un `DataSet`.

```
SqlCommand comando = connexion.CreateCommand();
comando.CommandText = "SELECT * FROM Alumnos";
adaptador.SelectCommand = comando;
```

Llegados a este punto solo queda llenar un `DataSet` gracias al método `Fill` del `SqlDataAdapter`. Este método recibe como parámetro (en sus versiones más sencillas) un `DataSet` o un `DataTable`. A continuación instanciaremos un `DataSet` que pasaremos a nuestro método.

```
DataSet ds = new DataSet();
adaptador.Fill(ds);
```

Es posible conocer el número de filas agregadas o actualizadas en el `DataSet` recuperando el valor que devuelve la llamada al método `Fill`.

```
int numFilas = adaptador.Fill(ds);
```

La primera llamada al método `Fill` crea un elemento en la colección `ds.Tables`. Este elemento es de tipo `DataTable` y contiene todos los registros devueltos por el comando de selección. El nombre de este objeto `DataTable` (propiedad `TableName`) es "Table", que no resulta especialmente explícito. Cada nueva tabla creada de esta manera mediante el método `Fill` tendrá como nombre *Table*, *Table1*, *Table2*, etc. Para evitar esta nomenclatura poco clara, se utiliza el método `Fill` pasándole el nombre del `DataTable` que debe rellenar:

```
adaptador.Fill(ds, "Alumnos");
```

La conexión desde la que se crea el comando puede abrirse o cerrarse, en ambos casos, todo funcionará bien. La diferencia reside en el comportamiento de cara a esta conexión:

- Si la conexión no está abierta en el momento de la ejecución, el método la abrirá, llenará el `DataSet` y, a continuación, la cerrará.
- Si la conexión está previamente abierta, la ejecución del método únicamente llenará el `DataSet`, sin modificar el estado de la conexión.

Cuando el `SqlDataAdapter` construye un `DataTable`, utiliza los nombres de los campos devueltos para nombrar cada una de sus `DataColumn`. Para definir los nombres de los campos es posible crear objetos de tipo `DataTableMapping` y, a continuación, agregarlos a la colección `TableMappings` del `SqlDataAdapter`.

La clase DataTableMapping provee las propiedades SourceTable y DataSetTable que permiten mapear un nombre de tabla en base de datos a un nombre de DataTable. Este tipo contiene, también, una colección ColumnMappings que hay que llenar con objetos de tipo DataColumnMapping cuyas propiedades SourceColumn y DataSetColumn permitan, respectivamente, especificar el nombre del campo en la base de datos y el nombre del campo asociado en el DataTable.

Para aplicar esta información de mapeo, se utiliza el método Fill pasándole como segundo parámetro el nombre de la tabla en base de datos.

El siguiente código crea un mapeo para las columnas *Identificador* y *FechaNacimiento* y llena un objeto DataTable llamado *DataTableAlumnos* con los datos de la tabla *Alumnos*. A continuación, muestra por pantalla el nombre de las columnas del DataTable.

```
var conexión = new
SqlConnection("Server=(LocalDb)\\v11.0;Initial
Catalog=Test;Integrated Security=True");
SqlCommand comando = connexion.CreateCommand();
comando.CommandText = "SELECT * From  Alumnos;";

SqlDataAdapter adaptador = new SqlDataAdapter();
adaptador.SelectCommand = comando;

DataTableMapping mapeoTabla = new DataTableMapping("Alumnos",
"DataTableAlumnos");
mapeoTabla.ColumnMappings.Add(new
DataColumnMapping("Identificador", "Id"));
mapeoTabla.ColumnMappings.Add(new
DataColumnMapping("FechaNacimiento", "Date"));
adaptador.TableMappings.Add(mapeoTabla);

DataSet ds = new DataSet();
adaptador.Fill(ds, "Alumnos");

foreach (DataColumn column in
ds.Tables["DataTableAlumnos"].Columns)
{
    Console.WriteLine(column.ColumnName);
}
```

Replicar las restricciones de la base de datos en un DataSet

El uso del método Fill permite recuperar los datos desde un origen de datos y los utiliza para llenar un DataSet. Pero el comportamiento por defecto de este método no incluye ninguna transferencia de información del esquema, como por ejemplo las claves primarias.

Es interesante poder recuperar las restricciones de las claves primarias cuando los datos que forman el DataSet deben actualizarse a partir de la base de datos. Sin estas restricciones, cada ejecución del método Fill agregará nuevas filas al DataSet, lo cual puede entrañar la presencia de datos duplicados. En el caso contrario, si el valor de la clave primaria de un registro de la base de datos se encuentra en el DataSet, el registro no se agrega a este DataSet: la fila correspondiente se actualiza con los datos frescos.

Este mecanismo de transferencia de la estructura de una tabla se implementa mediante el método FillSchema

de la clase SqlDataAdapter. Este método debe invocarse antes de realizar la primera llamada al método Fill.

```
adaptador.FillSchema(ds, SchemaType.Mapped);
adaptador.Fill(ds, "Alumnos");
```

c. Llenar un DataSet sin base de datos

Si bien la mayoría de aplicaciones utilizan una base de datos para realizar la persistencia de la información que procesan, otras utilizan información proveniente de la Web, de archivos de texto, o incluso generan sus propios datos. Estas aplicaciones también pueden utilizar las clases DataSet y DataTable. En cambio, no pueden utilizar el tipo SqlDataAdapter, lo que implica que la creación de la estructura de datos queda, por completo, a cargo del desarrollador.

Para crear una tabla en memoria, la primera operación que debemos realizar es la creación de un objeto DataTable. Su constructor recibe como parámetro el nombre de la tabla.

```
DataTable tabla = new DataTable("Alumnos");
```

A continuación, se crean las distintas columnas de la tabla proporcionando objetos de tipo DataColumn a la propiedad Columns del DataTable. El constructor de la clase DataColumn que nos interesa aquí recibe dos parámetros: el nombre de la columna y el tipo de datos que debe almacenar.

Para especificar el tipo, resulta interesante utilizar el operador typeof. Este recibe como parámetro un nombre de tipo .NET y devuelve un objeto de tipo System.Type.

```
DataColumn columnaIdentificador = new DataColumn("Identificador",
typeof(int));
DataColumn columnaNombre = new DataColumn("Nombre", typeof(string));
DataColumn columnaApellidos = new DataColumn("Apellidos", typeof(string));
DataColumn columnaFechaNacimiento = new DataColumn("FechaNacimiento",
typeof(DateTime));

tabla.Columns.Add(columnaIdentificador);
tabla.Columns.Add(columnaNombre);
tabla.Columns.Add(columnaApellidos);
tabla.Columns.Add(columnaFechaNacimiento);
```

Puede resultar útil precisar que cada registro debe tener un valor diferente para la columna *Identificador*. Para ello, existen dos soluciones a disposición del desarrollador.

Es posible asignar valor a la propiedad Unique del objeto DataColumn a true. De este modo, cada vez que se agrega un dato, se realiza una verificación sobre el valor del identificador proporcionado. Si ya existe, se produce una excepción.

Esta solución requiere que el desarrollador calcule él mismo el valor del identificador para cada registro, lo cual supone una mayor flexibilidad, pero también una mayor complejidad.

Otra solución consiste en asignar valor a la propiedad AutoIncrement del objeto DataColumn a true. De este modo, cada vez que se agregue un dato, el objeto DataTable gestionará el cálculo del identificador. Es

posible editar dos campos para modificar el cálculo: la propiedad AutoIncrementSeed, que indica el valor que debe utilizarse para el primer registro, y la propiedad AutoIncrementValue, que define el valor del incremento que debe utilizarse para realizar el cálculo.

Por definición, debe informarse en cualquier caso un identificador y es posible utilizar la propiedad AllowDBNull para indicar si los valores almacenados en esta columna pueden o no ser iguales a DBNull.Value (el equivalente en .NET al valor NULL de SQL).

La creación de la variable columnaIdentificador se realiza de la siguiente manera:

```
DataColumn columnaIdentificador = new DataColumn("Identificador",
typeof(int));
columnaIdentificador.Unique = true;
columnaIdentificador.AutoIncrement = true;
//AutoIncrementSeed vale por defecto 0 y AutoIncrementValue
//vale por defecto 1
columnaIdentificador.AllowDBNull = false;
```

Desde un punto de vista estructural, parece lógico utilizar la columna *Identificador* como clave primaria para el DataTable. La propiedad PrimaryKey de la clase DataTable recibe un valor de tipo DataColumn[] que define la lista de columnas que componen la clave primaria.

```
tabla.PrimaryKey = new DataColumn[] { columnaIdentificador };
```

Ciertas propiedades de las columnas correspondientes se modifican automáticamente mediante esta asignación. Su propiedad AllowDBNull pasa a valer true, y la propiedad Unique pasa también a true si la clave primaria está compuesta por un único campo. En caso contrario, el valor de la propiedad no se modifica.

Una vez creada esta estructura de tabla, tan solo queda insertar los datos. Este llenado se realiza mediante objetos de tipo DataRow, creados mediante el método NewRow del DataTable. Este método devuelve un DataRow que respeta el esquema definido para la tabla. Los datos de cada DataRow se almacenan en su propiedad ItemArray, de tipo object[]. Conviene asociar cada uno de los datos a almacenar con el elemento correspondiente de esta tabla.

Existen dos posibilidades para acceder a cada uno de los elementos:

- Acceder al índice correspondiente mediante el indexador de la clase DataRow:

```
fila[0] = <valor>;
```

- Acceder al elemento a través del indexador de la clase DataRow que recibe un nombre de columna:

```
fila["<Nombre de columna>"] = <valor>;
```

Cada registro debe, a continuación, agregarse a la colección Rows del objeto DataTable.

El siguiente código utiliza los dos tipos de acceso descritos más arriba para asociar los datos y agrega en la tabla los registros creados.

```
DataRow row1 = tabla.NewRow();
```

```
row1[1] = "Enrique";
row1[2] = "MARTIN";
row1[3] = new DateTime(1954, 01, 25);
tabla.Rows.Add(row1);

DataRow row2 = tabla.NewRow();
row2[1] = "Eric";
row2[2] =  "DIAZ";
row2[3] = new DateTime(1982, 05, 04);
tabla.Rows.Add(row2);

DataRow row3 = tabla.NewRow();
row3["Nombre"] = "Miguel";
row3["Apellidos"] = "MARTIN";
row3["FechaNacimiento"] = new DateTime(2004, 05, 28);
tabla.Rows.Add(row3);
```

2. Manipulación de datos sin conexión

Aunque un `DataSet` se haya llenado a partir de registros provenientes de una base de datos o manualmente a partir de datos generados por la aplicación, resulta esencial para el desarrollador poder manipular su contenido. Para ello, la clase `DataSet` y las clases asociadas disponen de muchas propiedades y numerosos métodos que facilitan la lectura, la edición o incluso la ordenación de los datos.

a. Lectura de datos

La clase `DataTable` provee distintas maneras de acceder a los datos que contiene. La más sencilla consiste en utilizar su colección `Rows`. Es posible tratar cada uno de los elementos almacenados en esta colección utilizando un bloque `foreach`.

```
foreach (DataRow row in tabla.Rows)
{
    Console.WriteLine("El identificador del registro es
{0}", row.ItemArray[0]);
}
```

La lectura de cada elemento del registro a partir de un índice puede resultar una operación algo delicada, en particular si se modifica el orden de las columnas de la tabla. El uso del método de acceso basado en el nombre de la columna es, naturalmente, mucho mejor.

```
foreach (DataRow row in tabla.Rows)
{
    Console.WriteLine("El identificador del registro es
{0}", row["Identificador"]);
}
```

Un último medio para leer los datos de un objeto `DataTable` consiste en crear un `DataTableReader`, que es un tipo derivado de `DbDataReader` y que provee varios métodos especializados para la lectura de datos con la gestión de un tipado fuerte. La clase `DataTableReader` se utiliza de la misma manera que el tipo

SqlDataReader (consulte la sección Selección de datos).

```
DataTableReader dataTableReader = tabla.CreateDataReader();
while (dataTableReader.Read())
{
    Console.WriteLine("El identificador del registro es
{0}", dataTableReader[0]);
}
```

b. Creación de restricciones

Las restricciones representan una parte importante en la gestión de los datos. Es habitual, en efecto, tener que aplicar ciertas restricciones sobre los juegos de datos para mantener su coherencia.

El framework .NET dispone de dos tipos de restricciones que podemos aplicar sobre los objetos DataTable: UniqueConstraint y ForeignKey Constraint.

UniqueConstraint

Esta restricción fuerza la unicidad de los valores de un grupo de DataColumn: si la restricción agrupa a tres columnas, la combinación de estos tres valores será siempre única. El hecho de no respetar esta restricción supone que se produzca una excepción de tipo System.Data.ConstraintException.

La creación de esta restricción se realiza instanciando un objeto de tipo UniqueConstraint y pasándole como parámetro un DataColumn o una tabla de objetos DataColumn. Es necesario, a continuación, agregarla a la colección Constraints del DataTable.

```
UniqueConstraint restricción = new UniqueConstraint();new
DataColumn[] { columnaNombre, columnaApellidos }));

tabla.Constraints.Add(restricción);
```

Cuando la restricción debe definirse sobre un único DataColumn, puede ser mucho más sencillo poner su propiedad Unique a true para obtener el mismo resultado.

ForeignKeyConstraint

Las ForeignKeyConstraint (o restricciones de claves foráneas) definen el comportamiento de las tablas vinculadas en caso de modificación o de eliminación de algún dato en el DataTable principal. Las propiedades UpdateRule y DeleteRule de este tipo permiten definir el tipo de acción que se desea realizar mediante un valor enumerado del tipo System.Data.Rule:

- None: las filas asociadas al dato editado o eliminado no se ven afectadas.
- Cascade: todas las filas relacionadas se modifican o eliminan.
- SetDefault: para cada fila asociada, el dato se remplaza por el valor por defecto de la columna (propiedad DefaultValue del DataColumn).
- SetNull: tras la aplicación de este comportamiento, el valor de los datos de la columna implicada es NULL para cada una de las filas.

La creación de una restricción de este tipo requiere la existencia de dos tablas. El primer parámetro del constructor de la clase `ForeignKeyConstraint` se corresponde con el nombre de la columna en la tabla principal, mientras que el segundo es el nombre de la columna asociada en una tabla vinculada. Los datos almacenados en ambas columnas deben ser del mismo tipo.

El código utilizado para los ejemplos anteriores se ha adaptado. Se crea, ahora, un `DataTable` suplementario llamado *CursoMúsica* y se le agrega una columna *FK_Curso* a la tabla *Alumnos* con los datos suplementarios. Se define y se agrega una restricción de clave a esta última tabla para referenciar correctamente al identificador de un curso de música para cada registro de la tabla. Aquí, la tabla principal es *CursoMúsica* y la tabla vinculada es *Alumnos*.

```
//Creación de la tabla CursoMúsica
DataTable tablaCurso = new DataTable("CursoMúsica");

DataColumn columnaIdentificadorCurso = new
DataColumn("IdentificadorCurso", typeof(int));
columnaIdentificadorCurso.AutoIncrement = true;
DataColumn columnaTítulo = new DataColumn("Título",
typeof(string));
tablaCurso.Columns.Add(columnaIdentificadorCurso);
tablaCurso.Columns.Add(columnaTítulo);

tablaCurso.PrimaryKey = new DataColumn[] {
columnaIdentificadorCurso };

DataRow rowCurso1 = tablaCurso.NewRow();
rowCurso1["Título"] = "Piano - Iniciación";
tablaCurso.Rows.Add(rowCurso1);

DataRow rowCurso2 = tablaCurso.NewRow();
rowCurso2 ["Título"] = "Guitarra - Iniciación";
tablaCurso.Rows.Add(rowCurso2);

DataRow rowCurso3 = tablaCurso.NewRow();
rowCurso3 ["Título"] = "Solfeo";
tablaCurso.Rows.Add(rowCurso3);

//Creación de la tabla Alumnos
DataTable tabla = new DataTable("Alumnos");

DataColumn columnaIdentificador = new DataColumn("Identificador",
typeof(int));
columnaIdentificador.AutoIncrement = true;
DataColumn columnaNombre = new DataColumn("Nombre",
typeof(string));
DataColumn columnaApellidos = new DataColumn("Apellidos",
typeof(string));
DataColumn columnaFechaNacimiento = new
DataColumn("FechaNacimiento",
typeof(DateTime));
DataColumn columnaFKCurso = new DataColumn("FK_Curso",
typeof(int));
```

```
tabla.Columns.Add(columnaIdentificador);
tabla.Columns.Add(columnaNombre);
tabla.Columns.Add(columnaApellidos);
tabla.Columns.Add(columnaFechaNacimiento);
tabla.Columns.Add(columnaFKCurso);
tabla.PrimaryKey = new DataColumn[] { columnaIdentificador };

//Creación de la restricción ForeignKeyConstraint
ForeignKeyConstraint restricciónFK = new
ForeignKeyConstraint(columnaIdentificadorCurso, columnaFKCurso);
restricciónFK.UpdateRule =  Rule.Cascade;
restricciónFK.DeleteRule = Rule.SetNull;
tabla.Constraints.Add(restricciónFK);

DataRow row1 = tabla.NewRow();
row1[1] = "Enrique";
row1[2] = "MARTIN FERNANDEZ";
row1[3] = new DateTime(1954, 01, 25);
row1[4] = tablaCurso.Rows[2]["IdentificadorCurso"];
tabla.Rows.Add(row1);

DataRow row2 = tabla.NewRow();
row2[1] = "Eric";
row2[2] = "DIAZ CASADO";
row2[3] = new DateTime(1982, 05, 04);
row2[4] = tablaCurso.Rows[2]["IdentificadorCurso"];
tabla.Rows.Add(row2);

DataRow row3 = tabla.NewRow();
row3["Nombre"] = "Miguel";
row3["Apellidos"] = "MARTIN  ROJO";
row3["FechaNacimiento"] = new DateTime(2004, 05, 28);
row3[4] = tablaCurso.Rows[1]["IdentificadorCurso"];
tabla.Rows.Add(row3);
```

c. Relaciones entre DataTables

Las relaciones entre las tablas son un elemento importante de los `DataSet`. Como las relaciones por clave foránea en las bases de datos, permiten navegar entre los distintos objetos `DataTable` que constituyen un almacenamiento de datos local. Se utilizan, a menudo, junto a las restricciones pues son complementarias en cuanto a su uso.

Crear una relación

Las relaciones se representan mediante objetos de la clase `DataRelation`. La creación de una relación se realiza pasando los objetos `DataColumn` padre e hijo al constructor del tipo. Es posible, también, especificar el nombre de la relación para hacer referencia a ella más fácilmente. Las distintas relaciones creadas para un `DataSet` deben agregarse a la colección `Relations` de este contenedor.

El siguiente código agrega los `DataTable` *Alumnos* y *CursoMúsica* a un `DataSet`, a continuación crea una relación llamada *Relación_Curso_Alumno* entre el campo *IdentificadorCurso* de la tabla *CursoMúsica* y el campo *FK_Curso* de la tabla *Alumnos*.

```
DataSet ds = new DataSet();
ds.Tables.Add(tabla);
ds.Tables.Add(tablaCurso);

ds.Relations.Add("Relacion_Curso_Alumno", columnaIdentificadorCurso,
columnaFKCurso);
```

Recorrer las relaciones

El uso de relaciones en un `DataSet` permite obtener todos los registros de un `DataTable` vinculados a un `DataRow` de otro objeto `DataTable`. El método `GetChildRows` de la clase `DataRow` realiza esta operación y devuelve los registros en una tabla de `DataRow`. Recibe como parámetro el nombre de la relación que se utilizará para seguir el vínculo que representa.

El siguiente código enumera los distintos cursos de música existentes junto al número de alumnos inscritos en cada uno.

```
foreach (DataRow filaCurso in ds.Tables["CursoMúsica"].Rows)
{
    Console.WriteLine("Los siguientes alumnos se han inscrito en el curso
{0}", filaCurso["Título"]);

    DataRow[] filaAlumnos =
filaCurso.GetChildRows("Relación_Curso_Alumno");

    if (filaAlumnos.Length > 0)
    {
        foreach (DataRow filaAlumno in filaAlumnos)
        {
            Console.WriteLine("\t{0} {1}", filaAlumno["Nombre"],
filaAlumno["Apellidos"]);
        }
    }
    else
    {
        Console.WriteLine("----------------  Ninguno -----------------");
    }
}
```

El resultado de la ejecución es el siguiente:

```
Los siguientes alumnos se han inscrito en el curso Piano - Iniciación
----------------- Ninguno ------------------
Los siguientes alumnos se han inscrito en el curso
Guitarra - Iniciación
        Miguel MARTIN ROJO
Los siguientes alumnos se han inscrito en el curso Solfeo
        Enrique MARTIN FERNANDEZ
        Eric DIAZ CASADO
```

d. Estado y versiones de un DataRow

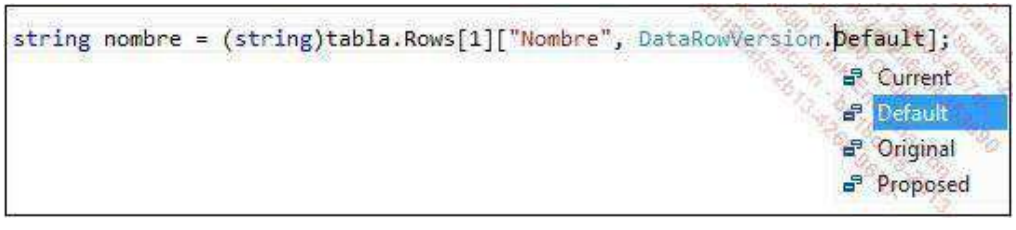
El tipo `DataRow` es capaz de seguir las modificaciones realizadas sobre su contenido. Su estado se almacena en la propiedad `RowState`, que contiene un valor de la enumeración `System.Data.DataRowState`. Los distintos estados posibles para un `DataRow` son cinco en total:

- `Detached`: la fila no está asociada a un `DataTable`.
- `Unchanged`: la fila no se ha modificado tras la llamada al método `Fill` o desde la última llamada a su método `AcceptChanges`.
- `Added`: la fila se ha agregado a la colección de registros de un `DataTable` pero todavía no se ha realizado ninguna llamada a su método `AcceptChanges`.
- `Deleted`: el objeto `DataRow` se ha eliminado de la tabla a la que estaba asociado mediante su método `Delete`.
- `Modified`: al menos un dato del registro se ha modificado y su método `AcceptChanges` no se ha invocado todavía.

Es posible también acceder a las distintas versiones de una fila mediante cuatro valores de la enumeración `DataRowVersion`.

- `Original`: versión original de la fila. Cuando se validan las modificaciones mediante el método `AcceptChanges`, la versión `Original` se modifica para reflejar el estado tras la llamada a este método. Esta versión no existe cuando el estado del registro es `Added`.
- `Current`: estado actual de la fila. Cuando el registro está en estado `Deleted`, esta versión no existe.
- `Proposed`: cuando la fila está siendo modificada, esta versión contiene los datos modificados, pero todavía no validados. Al finalizar la edición, el valor de esta versión se asocia con la versión `Current`. Los registros que no están vinculados con un `DataTable` (estado `Detached`) poseen, también, una versión `Proposed`.
- `Default`: es el estado por defecto de una fila. Cuando el estado del registro es `Added`, `Modified` o `Deleted`, esta versión es equivalente a la versión `Current`. Si el estado es `Detached`, equivale a la versión `Proposed`.

El acceso a un dato de una versión específica se realiza mediante el indexador implementado en la clase `DataRow`. Además del índice o del nombre de la columna, es posible especificar la versión del dato a manipular.



El método `HasVersion` de la clase `DataRow` permite saber si un registro posee un tipo de versión concreto. Recibe como parámetro el valor de la enumeración que identifica a la versión que se desea buscar y devuelve un valor booleano. Su uso permite evitar acceder a una versión del registro que no existe.

```
if (tabla.Rows[1].HasVersion(DataRowVersion.Original))
{
    string nombreAlumno = (string)tabla.Rows[1]["Nombre",
DataRowVersion.Original];
```

```
    //Continuamos el  procesamiento
}
```

e. Modificación de datos

La modificación de los datos de un registro se realiza asignando valores a sus distintos campos. Cada modificación se registra en la versión `Current` de la fila y su estado pasa a `Modified`. Cuando se desea realizar varias modificaciones, pueden aparecer problemas de coherencia. La edición de una columna que forma parte de una restricción de unicidad puede generar un valor duplicado temporalmente, lo cual produce una excepción incluso aunque se esté teniendo en cuenta la modificación. Este tipo de problema puede evitarse pasando la fila que estamos modificando al modo de edición.

Esta acción se realiza invocando al método `BeginEdit` del objeto `DataRow`, lo cual deshabilita temporalmente las verificaciones de las restricciones sobre los valores que contiene. Cualquier modificación realizada en este modo se almacena en la versión `Proposed` del `DataRow`. Una vez terminada la manipulación de los datos del registro, la validación de las modificaciones se lleva a cabo con una llamada a `EndEdit`. Es posible, también, anular los cambios realizados mediante el método `CancelEdit`.

A continuación se muestra un ejemplo de código que remplaza el identificador de una fila por el identificador de otra fila. La excepción no se produce mientras se realiza la edición sino tras la llamada al método `EndEdit`: la restricción de unicidad de la clave primaria se deshabilita durante la edición.

```
DataRow filaAEditar = tabla.Rows[1];
filaAEditar.BeginEdit();
//Sin el modo de edición, la siguiente línea produce una excepción
filaAEditar["Identificador"] = tabla.Rows[2]["Identificador"];
//pero en modo edición, la excepción se produce
//tras la llamada a EndEdit()
filaAEditar.EndEdit();
```

El siguiente código modifica un nombre y valida la modificación.

```
DataRow filaAEditar = tabla.Rows[2];
Console.WriteLine("El nombre del alumno es {0}",
filaAEditar["Nombre"]);

filaAEditar.BeginEdit();
filaAEditar["Nombre"] = "Miguel";
filaAEditar.EndEdit();

Console.WriteLine("El nombre del alumno es, ahora, {0}",
filaAEditar["Nombre"]);
```

La ejecución de este último ejemplo produce el siguiente resultado:

```
El nombre del alumno es Miguel
El nombre del alumno es, ahora, Miquel
```

f. Eliminación de datos

Existen dos soluciones para hacer que una fila no exista más en un `DataTable`. Ambas tienen un comportamiento ligeramente distinto.

La eliminación pura y simple de un registro se realiza mediante el método `Remove` de la clase `DataRowCollection`. Recibe como parámetro el `DataRow` que se desea eliminar.

El siguiente código muestra cómo eliminar la primera fila de un `DataTable` utilizando este método.

```
tabla.Rows.Remove(tabla.Rows[0]);
```

La segunda solución es menos radical: marca un registro como eliminado pero no lo borra hasta que se validen las modificaciones en el objeto `DataTable`. Si las modificaciones se anulan, la fila se queda en el `DataTable` y recupera su estado anterior. Para realizar esta acción, se utiliza el método `Delete` del objeto `DataRow` que se desea eliminar.

```
tabla.Rows[0].Delete();
```

El uso del método `Delete` sobre un registro en el estado `Added` elimina definitivamente el registro, pues no tiene versión original.

g. Validar o anular las modificaciones

La validación de modificaciones mediante el método `EndEdit` del objeto `DataRow` realiza una validación reversible: es posible volver a la versión anterior mediante las distintas versiones de cada una de las filas.

El método `AcceptChanges` permite validar definitivamente las modificaciones **locales** de un `DataRow`, de un `DataTable` o de un `DataSet`. Este método se implementa, efectivamente, en cada uno de los tipos, lo que ofrece cierta flexibilidad.

Su funcionamiento es el siguiente:

- Invoca al método `EndEdit` de cada una de las filas en modo edición.
- Para cada fila en el estado `Added` o `Modified`, copia los valores de la versión `Current` a la versión `Original`. El estado de estas filas es, ahora, `Unchanged`.
- Cada registro cuyo estado es `Deleted` se elimina definitivamente.

El método `RejectChanges` anula definitivamente cada una de las modificaciones locales realizadas. Como ocurre con el método `AcceptChanges`, se implementa en los objetos `DataRow`, `DataTable` y `DataSet`. Procesa los registros de una manera completamente opuesta al método `AcceptChanges`:

- Ejecuta el método `CancelEdit` de cada una de las filas.
- Modifica cada una de las filas en estado `Deleted` o `Modified` de manera que vuelvan al estado `Unchanged` y los datos de su versión `Original` y `Current` sean idénticos.
- Los registros en el estado `Added` se eliminan definitivamente.

Cabe destacar que las restricciones `ForeignKeyConstraint` poseen una propiedad `AcceptRejectRule` que permite recuperar las modificaciones realizadas en sus filas hijas cuando se invoca a los métodos `AcceptChanges` o `RejectChanges`.

h. Filtrado y ordenación mediante un DataView

Es posible modificar la manera en la que se ve la información de un `DataTable` para modificar los registros subyacentes. Para ello, desde un punto de vista orientado a la base de datos, se utilizan las vistas. Estas tienen, también, un equivalente en .NET: los `DataView`. Permiten filtrar u ordenar los elementos de un `DataTable` sin tener que recurrir al envío de consultas SQL que contengan cláusulas `WHERE` u `ORDER BY` a la base de datos. El procesamiento que permite crear estas vistas es totalmente local y se basa en un objeto `DataTable` que contiene los datos. Pueden coexistir varios `DataView` para el mismo `DataTable`, cada uno correspondiente a un punto de vista diferente sobre los mismos datos.

El origen de datos local de tipo `DataTable` puede pasarse como parámetro al constructor de la clase `DataView` o asociarse a su propiedad `Table`. Las propiedades `RowFilter` y `RowStateFilter` de la vista filtran los registros en función de predicados o de su estado. La propiedad `Sort` permite precisar los campos en función de los cuales se ordenarán los registros.

Los registros presentes en un `DataView` se presentan bajo la forma de objetos de tipo `DataRowView`.

RowFilter

El filtro se define según una cadena de caracteres que indica el predicado que debe respetarse para que las filas estén disponibles en el `DataView`. La sintaxis general de esta cadena de caracteres es parecida a la de la cláusula `WHERE` de una consulta SQL y puede ser la combinación de varias condiciones gracias a las palabras clave `AND` y `OR`.

Para obtener la lista de alumnos cuyo apellido es Martín, podemos utilizar el siguiente código.

```
DataView vistaFiltrada = new DataView(tabla);
vistaFiltrada.RowFilter = "Apellido = 'MARTIN'";

foreach (DataRowView fila in vistaFiltrada)
{
    Console.WriteLine("{0} {1}", fila["Apellidos"], fila["Nombre"]);
}
```

RowStateFilter

Mediante esta propiedad es posible precisar el estado esperado de los registros filtrados, así como la versión que debe estar visible en el `DataView`.

Esta propiedad es un valor del tipo enumerado `DataViewRowState`. Los valores que expone este tipo se describen a continuación:

- `None`: ninguna fila.
- `Unchanged`: filas no modificadas.
- `Added`: filas agregadas.

- `Deleted`: filas marcadas como eliminadas.
- `ModifiedCurrent`: filas modificadas, versiones actuales de los datos.
- `CurrentRows`: filas de la versión actual (no modificadas, nuevas y modificadas). Es el equivalente a la combinación de los valores `Unchanged`, `Added` y `ModifiedCurrent`.
- `ModifiedOriginal`: versión original de las filas que se han modificado.
- `OriginalRows`: versión original de las filas que no se han agregado. Es el equivalente a la combinación de los valores `ModifiedOriginal`, `Deleted` y `Unchanged`.

```
DataView vistaFiltradaPorEstado = new DataView(tabla);
vistaFiltradaPorEstado.RowStateFilter = DataViewRowState.Deleted;

Console.WriteLine("Los alumnos eliminados son:");
foreach (DataRowView fila in  vistaFiltradaPorEstado)
{
    Console.WriteLine("\t{0} {1}", fila["Apellidos"],
fila["Nombre"]);
}
```

La enumeración `DataViewRowState` posee el atributo `Flags`, lo cual significa que es posible combinar varios valores para formar uno nuevo. El valor que resulta de esta combinación permite, de este modo, extender los criterios de búsqueda y obtener así más resultados.

Esta operación se realiza mediante el operador **O lógico** representado por el símbolo |.

```
vistaFiltradaPorEstado.RowStateFilter = DataViewRowState.Deleted |
DataViewRowState.Added;
```

La combinación anterior permite visualizar en el `DataView` las filas cuyo estado es `Deleted` o `Added`.

Sort

Como con el filtrado, la ordenación requiere que se asigne valor a una propiedad (`Sort`) con una cadena de caracteres. Esta última es parecida a la cláusula `ORDER BY` de una consulta SQL. Es posible, de este modo, ordenar los distintos alumnos por orden alfabético de la siguiente manera:

```
DataView vistaOrdenada = new DataView(tabla);
vistaOrdenada.Sort = "Nombre ASC, Apellidos ASC";

foreach (DataRowView fila in vistaOrdenada)
{
    Console.WriteLine("{0} {1}", fila["Nombre"], fila["Apellidos"]);
}
```

i. Búsqueda de datos

La búsqueda en un juego de datos se realiza mediante varios métodos implementados en las clases `DataTable` y `DataView`. Estos métodos se corresponden con dos modos de búsqueda distintos: por predicado o por clave.

Búsqueda por predicado

Este tipo de búsqueda se corresponde con la ejecución de una consulta `SELECT` en una base de datos. El método en el que se implementa se denomina, además, `Select` y se define en el tipo `DataTable`. Recibe como parámetros hasta dos cadenas de caracteres y un valor de enumeración `DataViewRowState` que representan, respectivamente, un filtro sobre los registros, una clase de ordenación y el estado de las filas esperadas. Estos distintos parámetros reciben la misma forma que las propiedades `RowFilter`, `RowStateFilter` y `Sort` del tipo `DataView`.

Esta operación de búsqueda devuelve una tabla de objetos `DataRow` que contienen los registros correspondientes a los criterios definidos en la llamada.

```
DataRow[] losMartín = tabla.Select("Apellidos = 'Martín'", "Nombre
ASC");
Console.WriteLine("Los alumnos cuyo apellido es MARTIN son:");
foreach (DataRow fila in losMartín)
{
    Console.WriteLine("\t{0} {1}", fila["Apellidos"],
fila["Nombre"]);
}
```

Búsqueda por clave

Este tipo de búsqueda permite recuperar uno o varios elementos en un `DataView` ordenado mediante los métodos `Find` y `FindRows`. Es necesario que los registros estén ordenados, pues la búsqueda se realiza sobre la totalidad de los campos seleccionados en la propiedad `Sort` del `DataView`. Cada uno de los criterios proporcionados en la llamada al método se corresponde con la columna ordenada posicionada en el mismo índice.

Para aclarar este último punto, consideremos que el criterio de ordenación es el siguiente: " *Nombre ASC, Apellidos ASC*". La llamada a los métodos de búsqueda debe proporcionar un valor para el nombre y un valor para los apellidos, en este orden.

El método `Find` devuelve un valor numérico que es el índice de la primera fila que se corresponde con los criterios de búsqueda. Si no se encuentra ningún registro, se devuelve el valor `-1`.

```
DataView vistaOrdenadaPorNombre = new DataView(tabla);
vistaOrdenadaPorNombre.Sort = "Nombre ASC";

int índice = vistaOrdenadaPorNombre.Find(new[] { "Antonio" });
if (índice == -1)
{
    Console.WriteLine("Ningún alumno tiene el nombre Antonio");
}
else
{
    DataRowView fila = vistaOrdenadaPorNombre[índice];
    Console.WriteLine("El primer alumno cuyo nombre es Antonio
es {0} {1}", fila["Nombre"], fila["Apellidos"]);
}
```

El método FindRows devuelve una tabla de objetos DataRowView.

```
DataView vistaOrdenadaPorNombreApellidos = new DataView(table);
vistaOrdenadaPorNombreApellidos.Sort = "Nombre ASC, Apellidos ASC";

DataRowView[] filas = vistaOrdenadaPorNombreApellidos.FindRows(new[]
{ "Miguel", "MARTIN" });
if (filas.Length == 0)
{
    Console.WriteLine("Ningún alumno se llama Miguel MARTIN");
}
else
{
    Console.WriteLine("Hay {0} alumno(s) cuyo nombre es Miguel
MARTIN", filas.Length);
}
```

3. Validar las modificaciones en la base de datos

Hasta el momento, todas las modificaciones validadas lo estaban de manera local, a nivel de los DataRow, DataTable o DataSet. Pero cuando los datos de estos elementos provienen de una base de datos, es posible transferir las modificaciones realizadas de manera local hacia la base de datos automáticamente.

Para ello, se invoca al objeto SqlDataAdapter que se utilizó para llenar el DataSet con los registros provenientes de la base de datos.

El método Update de este SqlDataAdapter tiene como objetivo validar los datos que se le envían ejecutando consultas SQL de agregación, actualización o eliminación definidas respectivamente por las propiedades InsertCommand, UpdateCommand y DeleteCommand. Estos comandos se ejecutan en función del estado de los registros que se deben validar. Si el método Update requiere alguno de estos comandos pero no se ha asignado ningún valor, entonces se produce una excepción.

La validación de las modificaciones realizadas sobre un juego de registros puede realizarse sobre el conjunto de un DataSet, sobre un DataTable o sobre un número restringido de filas.

El siguiente código crea un DataSet y lo llena a partir de una base de datos. Tras las modificaciones de los datos, se ejecuta el método Update del SqlDataAdapter para validar en la base de datos el conjunto de modificaciones realizadas.

```
var conexión = new
SqlConnection("Server=(LocalDb)\\v11.0;Initial
Catalog=Test;Integrated Security=True");
SqlCommand comando = conexión.CreateCommand();
comando.CommandText = "SELECT * From  Alumnos;";

SqlDataAdapter adaptador = new SqlDataAdapter();
adaptador.SelectCommand = comando;

DataSet ds = new DataSet();
adaptador.Fill(ds);
```

```
//Sitúe aquí todas las modificaciones sobre los registros

adaptador.Update(ds);
```

Este código es perfectamente válido, aunque produce una excepción, pues las propiedades `InsertCommand`, `UpdateCommand` y `DeleteCommand` no poseen ningún valor.

a. Generar los comandos de actualización automáticamente

La clase `SqlCommandBuilder` se diseñó para generar las consultas `INSERT`, `UPDATE` y `DELETE` de un `SqlDataAdapter` a partir de la asignación de valor a su propiedad `SelectCommand`. Es necesario, por tanto, que esta propiedad posea algún valor. Para que el `SqlCommandBuilder` pueda generar comandos es preciso tener en cuenta otras dos restricciones: los objetos `DataTable` afectados por la actualización deben **tener una clave primaria** y sus datos no deben pertenecer a una unión entre varias tablas de bases de datos. Si no se respetan estas condiciones, se produce una excepción durante la generación de los comandos.

Tras la ejecución del método `Update`, los valores presentes en la base de datos deben corresponderse con la versión `Original` almacenada en el `DataSet`. Si no fuera el caso, se produciría una excepción del tipo `DBConcurrencyException`.


Veremos cómo evitar este tipo de excepciones en la sección Gestión de los accesos concurrentes, en este mismo capítulo.

Para utilizar el tipo `SqlCommandBuilder`, basta con instanciarlo pasándole como parámetro un `SqlDataAdapter` a partir del que se generarán los comandos. Para visualizar la consulta encapsulada en uno de estos comandos es preciso utilizar los métodos `GetInsertCommand`, `GetUpdateCommand` o `GetDeleteCommand`. Estos métodos devuelven todos ellos un objeto `DbCommand` cuya propiedad `CommandText` representa la consulta SQL asociada.

Este ejemplo parte del código anterior y agrega las instrucciones necesarias para generar y mostrar los comandos para la variable `adaptador`.

```
var conexión = new
SqlConnection("Server=(LocalDb)\\v11.0;Initial
Catalog=Test;Integrated Security=True");
SqlCommand comando = connexion.CreateCommand();
comando.CommandText = "SELECT * From  Alumnos;";

SqlDataAdapter adaptador = new SqlDataAdapter();
adaptador.SelectCommand = comando;

DataSet ds = new DataSet();
adaptador.Fill(ds);

//Sitúe aquí todas las modificaciones de registros

//Creación del SqlCommandBuilder y asociación al SqlDataAdapter
SqlCommandBuilder commandBuilder = new
SqlCommandBuilder(adaptador);

string consultaInsert =
commandBuilder.GetInsertCommand().CommandText;
```

```
string consultaUpdate =
commandBuilder.GetUpdateCommand().CommandText;
string consultaDelete =
commandBuilder.GetDeleteCommand().CommandText;

Console.WriteLine("Consulta INSERT:");
Console.WriteLine(consultaInsert);
Console.WriteLine();
Console.WriteLine("Consulta UPDATE:");
Console.WriteLine(consultaUpdate);
Console.WriteLine();
Console.WriteLine("Consulta DELETE:");
Console.WriteLine(consultaDelete);

adaptador.Update(ds);
```

Este ejemplo produce la siguiente visualización:

```
Consulta INSERT:
INSERT INTO [Alumnos] ([Nombre], [Apellidos], [FechaNacimiento]) VALUES
(@p1, @p2, @p3)

Consulta UPDATE:
UPDATE [Alumnos] SET [Nombre] = @p1, [Apellidos] = @p2, [FechaNacimiento] =
@p3 WHERE (([Identificador] = @p4) AND ([Nombre] = @p5) AND ([Apellidos] =
@p6) AND ((@p7 = 1 AND [FechaNacimiento] IS NULL) OR ([FechaNacimiento] =
@p8)))

Consulta DELETE:
DELETE FROM [Alumnos] WHERE (([Identificador] = @p1) AND ([Nombre] =
@p2) AND ([Apellidos] = @p3) AND ((@p4 = 1 AND [FechaNacimiento] IS
NULL) OR ([FechaNacimiento] = @p5)))
```

El objeto `SqlCommandBuilder` solo genera un comando si su valor es `null`. Dicho de otro modo, si un comando de inserción, de actualización o de eliminación ha recibido valor manualmente, no se sobrescribirá por un comando generado.

b. Comandos de actualización personalizados

La generación de comandos no es útil en todos los casos. Es bastante común, en efecto, en particular para aplicaciones empresariales, marcar ciertos registros como eliminados en lugar de eliminarlos definitivamente o guardar cierta información en una tabla de histórico tras cada modificación de la base de datos. Una solución para este caso consiste en definir comandos personalizados que permitan ejecutar consultas SQL adaptadas a las necesidades.

Para llevar a cabo esta acción basta con crear un comando y asignarlo a la propiedad correspondiente del objeto `SqlDataAdapter`.

c. Gestión de los accesos concurrentes

Las aplicaciones son cada vez con mayor frecuencia aplicaciones multiusuario, lo que implica que se producen cada

vez más accesos concurrentes. Existen dos puntos de vista para gestionar los conflictos que se producen: el bloqueo pesimista y el bloqueo optimista.

El bloqueo pesimista es el más restrictivo para los usuarios. Su modo de funcionamiento es muy sencillo: cuando un usuario manipula un registro, este se bloquea y no puede modificarlo ningún otro usuario. Este bloqueo no desaparece hasta que el usuario que trabaja con los datos valida sus modificaciones. Este modo de funcionamiento tiene como objetivo impedir cualquier conflicto.

En cuanto al bloqueo optimista, no se trata realmente de un bloqueo. Se corresponde más bien a un enfoque que aborda la resolución de conflictos antes que evitarlos. Los usuarios son, por tanto, libres a la hora de editar los datos de manera simultánea. Cuando se realice la validación efectiva de las actualizaciones, se realizará una comparación con los datos correspondientes en base de datos. Si la versión `Original` de los datos locales es diferente a los datos existentes en la base de datos, entonces quiere decir que otro usuario ha podido modificar estos datos y que, por tanto, se ha producido un conflicto.

Este enfoque se utiliza en los comandos de actualización y de eliminación generados automáticamente por el `SqlCommandBuilder`.

Existen tres soluciones para gestionar estas actualizaciones conflictivas:

- Descartar la actualización.
- Eliminar los datos de la base de datos con los datos más recientes.
- Pedir ayuda al usuario.

La primera solución es la utilizada por el objeto `SqlDataAdapter`. La ejecución de la consulta de actualización devuelve un valor entero que indica el número de registros modificados. Si este valor es igual a 0 cuando una fila modificada está siendo validada, el objeto `SqlDataAdapter` sabe que no se ha encontrado ninguna otra fila correspondiente a los criterios definidos por la consulta. Por defecto, se produce una excepción en este caso. Este comportamiento puede modificarse asignando el valor `true` a la propiedad `Continue UpdateOnError` del `SqlDataAdapter`.

Es posible controlar de manera más fina el comportamiento de la aplicación en caso de conflicto utilizando el evento `RowUpdated` de la clase `SqlDataAdapter`. Este evento, que se produce justo después de cada ejecución de una consulta de actualización, proporciona los elementos necesarios para analizar el resultado de la actualización y controlar la ejecución de las actualizaciones siguientes.

El delegado que define la firma del controlador del evento `RowUpdated` es el siguiente:

```
public delegate void SqlRowUpdatedEventHandler(object sender,
SqlRowUpdatedEventArgs e);
```

El objeto de tipo `SqlRowUpdatedEventArgs` que se pasa como parámetro al controlador de eventos posee una propiedad `Row` que permite conocer la fila que acaba de procesarse así como las dos propiedades `Command` y `StatementType` que contienen, respectivamente, el comando SQL ejecutado y un valor enumerado que representa el tipo de la consulta enviada a la base de datos.

Existen otras propiedades de este objeto que proporcionan una representación del resultado de la ejecución del comando. Es el caso de las propiedades `RowCount` y `RecordsAffected` que contienen, respectivamente, el número de filas que deben actualizarse y el número de registros afectados. La propiedad `Errors` contiene la excepción potencialmente producida tras la ejecución del comando.

Por último, la propiedad `UpdateStatus` permite conocer el comportamiento que va a adoptar la aplicación tras el

procesamiento del registro en curso y ofrece también la posibilidad al desarrollador de modificar esta ruta de ejecución, en particular en caso de error. Los valores disponibles para esta propiedad son los siguientes:

- `Continue`: la ejecución continúa.
- `ErrorsOccured`: se producen errores, es necesario procesarlos.
- `SkipCurrentRow`: la actualización de la fila en curso se abandona, pero la ejecución puede continuar para los demás registros.
- `SkipAllRemainingRows`: la fila en curso de procesamiento y todos los registros que vienen a continuación no deben actualizarse.

El siguiente código define un controlador de eventos que muestra el identificador de los registros cuya actualización no ha podido ejecutarse.

```
void adaptador_RowUpdated(object sender, SqlRowUpdatedEventArgs
args)
{
    if (args.RecordsAffected == 0)
    {
        Console.WriteLine("La fila con identificador {0}
no se ha actualizado.", args.Row["Identificador"]);
        args.Status = UpdateStatus.SkipCurrentRow;
    }
}
```

Utilizar las transacciones

Una transacción es un elemento que utiliza una base de datos para definir un conjunto de operaciones que deben ejecutarse de manera atómica. El grupo de consultas se considera como indivisible, de manera que si una de ellas falla, se anula la ejecución de la totalidad y la base de datos vuelve al estado anterior al inicio de la transacción. Esta **operación de anulación** se denomina, a menudo, **rollback**, mientras que la operación de **validación final** resultante de la ejecución correcta de cada una de las consultas se denomina **commit**.

Las transacciones se utilizan con frecuencia, por tanto, cuando existen varias consultas de modificación que deben realizarse pero el fallo de alguna de ellas supone una incoherencia de datos. Un ejemplo podría ser la gestión de las reservas en un avión o en un tren. Cuando todas las plazas están reservadas, las compañías aéreas mantienen una lista de espera para poder revender fácilmente una plaza cuando algún pasajero anula su reserva. En el momento de la anulación, nos encontramos con dos acciones a realizar:

- Anulación de la reserva del pasajero A para el vuelo 1234, asiento 45.
- Creación de la reserva para el cliente B, primero en la lista de espera, para el vuelo 1234, asiento 45.

Si la primera operación falla pero la segunda se produce correctamente, el estado de las reservas relativas al vuelo se vuelve incoherente. En efecto, dos pasajeros tendrían una reserva válida para el mismo vuelo y el mismo asiento, mientras que todos los demás asientos estarían ocupados, lo cual produciría, probablemente, problemas a los pasajeros y a la compañía aérea.

Con ADO.NET, se utiliza una transacción en la conexión que se utiliza para realizar los comandos que se desean agrupar. La llamada al método `BeginTransaction` de un objeto `SqlConnection` devuelve un objeto de tipo `SqlTransaction` que encapsula el uso de las transacciones en la base de datos. Esta transacción puede pasarse, a continuación, a los comandos mediante su propiedad `Transaction`. En caso de producirse algún error durante la ejecución de alguna de las consultas se invoca al método `Rollback` del objeto `SqlTransaction`. Cuando todas las consultas se han ejecutado con éxito, el método `Commit` del objeto `SqlTransaction` valida definitivamente las modificaciones realizadas sobre la base de datos.

```
private void UsoDeTransacción()
{
    SqlConnection conexión = new
SqlConnection("Server=(LocalDb)\\v11.0;Initial
Catalog=ReservasAvion;Integrated  Security=True");

    SqlTransaction transacción = conexión.BeginTransaction();

    try
    {
        SqlCommand comandoAnulación =
conexión.CreateCommand();
        comandoAnulación.CommandText = "DELETE FROM RESERVAS
WHERE NumeroVuelo=1234 AND NumeroAsiento=45";
        comandoAnulación.Transaction = transacción;

        SqlCommand comandoReserva =
conexión.CreateCommand();
        comandoAnulación.CommandText = "INSERT INTO RESERVAS
(NumeroVuelo, NumeroAsiento, IdentificadorCliente) VALUES (1234, 45,
4765169";
        comandoReserva.Transaction = transacción;
```

```
        comandoAnulación.ExecuteNonQuery();
        comandoReserva.ExecuteNonQuery();

        transacción.Commit();
    }
    catch (Exception ex)
    {
        transacción.Rollback();
    }

    conexión.Close();
}
```

Este código puede escribirse, también, de la siguiente manera utilizando estructuras `using`.

```
private static void UsoDeTransacción()
{
    using (SqlConnection conexión = new
SqlConnection("Server=(LocalDb)\\v11.0;Initial
Catalog=ReservasAvion;Integrated  Security=True"))
    using (SqlTransaction transacción =
conexión.BeginTransaction())
    {
        SqlCommand comandoAnulación = conexión.CreateCommand();
        comandoAnulación.CommandText = "DELETE FROM  RESERVAS
WHERE NumeroVuelo=1234 AND NumeroAsiento=45";
        comandoAnulación.Transaction = transacción;

        SqlCommand comandoReserva = conexión.CreateCommand();
        comandoAnulación.CommandText = "INSERT INTO  RESERVAS
(NumeroVuelo, NumeroAsiento, IdentificadorCliente) VALUES (1234, 45,
4765169";
        comandoReserva.Transaction = transacción;

        comandoAnulación.ExecuteNonQuery();
        comandoReserva.ExecuteNonQuery();

        transacción.Commit();
    }
}
```

En caso de producirse algún error, se invoca a la instrucción `Rollback` desde el método `Dispose` del objeto `SqlTransaction`. Dicha instrucción se invoca implícitamente mediante la estructura `using`.

Presentación de LINQ

LINQ (*Language Integrated Query*) está constituido por un conjunto de funcionalidades incluidas en el framework .NET a partir de la versión 3.5. Este componente está formado, por una parte, por una biblioteca de clases que proveen **capacidades de consulta** y, por otra, por un juego de **extensiones para el lenguaje C#** que permiten utilizar palabras clave específicas para escribir consultas con una sintaxis similar a la de **SQL**. Esta integración implica un cambio apreciable: la validez de una consulta se verifica en tiempo de compilación.

La extensión del lenguaje es, de hecho, un elemento sintáctico que simplifica la escritura de las consultas, aunque las expresiones que utilizan la sintaxis LINQ se traducen en el compilador por llamadas convencionales. Es posible deducir que cualquier consulta LINQ puede escribirse con llamadas a métodos. De manera inversa no ocurre lo mismo, algunos métodos no tienen su equivalencia en la sintaxis LINQ. No existe, en particular, ninguna palabra clave que permita traducir el método `FirstOrDefault` en la extensión LINQ de C#.

La existencia de ambos tipos de sintaxis presenta la ventaja de que deja al desarrollador escoger cuál utilizar en base a la que le resulte más cómoda. Hay quien prefiere escribir, en efecto, consultas de **manera fluida** utilizando únicamente las **palabras clave del lenguaje**, otros, entre los que me incluyo, tenemos una mayor afinidad por el uso de funciones y procedimientos.

LINQ es capaz de trabajar con **distintos orígenes de datos**, desde una **colección de objetos** local hasta **bases de datos SQL Server**, pasando por **archivos XML** o cualquier otro origen para el que exista un **proveedor LINQ**. La integridad de estos orígenes de datos puede consultarse mediante las **mismas consultas LINQ** gracias al trabajo realizado por los proveedores de datos. Un proveedor LINQ transforma los datos del origen correspondiente en objetos .NET y a la inversa. De este modo, las consultas LINQ trabajan sobre **objetos .NET fuertemente tipados**, la comunicación con los orígenes de datos es responsabilidad de los proveedores.

La estructura de LINQ permite extenderlo y existen distintas implementaciones de terceros de proveedores para numerosos orígenes de datos: **archivo CSV**, **directorios LDAP** o incluso bases de datos **Firebird** o **SQLite**.

Sintaxis

Antes de detallar los distintos elementos de LINQ que pueden utilizarse para escribir consultas, vamos a estudiar un primer ejemplo que nos permitirá comprender mejor qué es LINQ y cómo se utiliza.

Los datos sobre los que trabajaremos a lo largo de este estudio sintáctico de LINQ se encuentran en la base de datos Northwind de Microsoft. Una parte de su contenido se ha extraído en tres archivos en formato de texto que se cargarán en memoria para formar tres colecciones de objetos. Esta etapa de carga se realiza mediante el siguiente código, que se puede adaptar para apuntar a la carpeta que contenga los archivos. Los archivos de datos están disponibles para su descarga en el sitio web del fabricante.

```
class Cliente
{
    public string Identificador { get; set; }
    public string Nombre { get; set; }
    public string Direccion { get; set; }
    public string Ciudad { get; set; }
    public string Pais { get; set; }

    public List<ProductoPedido> ProductosPedidos { get; set; }
}

class ProductoPedido
{
    public int NumeroPedido { get; set; }

    //Referencia del Cliente asociado al pedido
    public string IdentificadorCliente { get; set;  }
    //Referencia del Comercial asociado al pedido
    public int IdentificadorComercial { get; set;  }

    public DateTime FechaPedido { get; set; }
    public string NombreProducto { get; set; }
    public decimal PrecioUnitario { get; set; }
    public int Cantidad { get; set; }
}

class Comercial
{
    public int Identificador { get; set; }
    public string Apellidos { get; set; }
    public string Nombre { get; set; }
}

class Program
{
    //Esta variable debe modificarse para que apunte
    //a la carpeta que contiene los archivos de datos
    private const string carpetaDatos = @"C:\DATOS\Ejemplos\";

    static void Main(string[] args)
    {
        Console.WriteLine("Carga de los clientes en memoria");
```

```
        List<Cliente> clientes = new List<Cliente>();
        List<ProductoPedido> productosPedidos =  new
List<ProductoPedido>();
        List<Comercial> comerciales = new List<Comercial>();

        //Lectura de los registros de ProductosPedidos.txt
        string[] filasProductosPedidos =
File.ReadAllLines(carpetaDatos + "ProductosPedidos.txt");
        foreach (string fila in  filasProductosPedidos)
        {
            ProductoPedido productoPedido =
new ProductoPedido();
            var valores = fila.Split('\t');

            productoPedido.NumeroPedido =
Convert.ToInt32(valores[0]);
            productoPedido.IdentificadorCliente = valores[1];
            productoPedido.IdentificadorComercial =
Convert.ToInt32(valores[2]);
            productoPedido.FechaPedido =
Convert.ToDateTime(valores[3]);
            productoPedido.NombreProducto = valores[4];
            productoPedido.PrecioUnitario =
Convert.ToDecimal(valores[5]);
            productoPedido.Cantidad =
Convert.ToInt32(valores[6]);

            productosPedidos.Add(productoPedido);
        }

        //Lectura de los registros de Clientes.txt
        string[] filasClientes =
File.ReadAllLines(destinationFolder + "Clientes.txt");
        foreach (string fila in filasClientes)
        {
            Cliente cliente = new Cliente();
            var valores = fila.Split('\t');

            cliente.Identificador = valores[0];
            cliente.Nombre = valores[1];
            cliente.Direccion = valores[2];
            cliente.Ciudad = valores[3];
            cliente.Pais = valores[4];

            cliente.ProductosPedidos = new
List<ProductoPedido>();
            foreach (ProductoPedido producto in
productosPedidos)
            {
                if (producto.IdentificadorCliente ==
cliente.Identificador)
                {
                    cliente.ProductosPedidos.Add(producto);
                }
```

```
            }

            clientes.Add(cliente);
        }

        //Lectura de los registros de Comerciales.txt
        string[] filasComerciales =
File.ReadAllLines(carpetaDatos + "Comerciales.txt");
        foreach (string fila in filasComerciales)
        {
            Comercial comercial = new Comercial();
            var valores = fila.Split('\t');

            comercial.Identificador = Convert.ToInt32(valores[0]);
            comercial.Apellidos = valores[1];
            comercial.Nombre = valores[2];

            comerciales.Add(comercial);
        }

        Console.WriteLine("Carga finalizada.");
        Console.ReadLine();
    }
}
```

1. Una primera consulta LINQ

Para escribir una consulta LINQ es preciso, en primer lugar, identificar un origen a partir del cual se podrán obtener los datos. Los orígenes de datos compatibles con LINQ son aquellas colecciones que implementan la interfaz `IEnumerable<T>`. Las listas de objetos cargados antes en memoria se almacenan en colecciones del tipo `List<T>`, que implementa, entre otras, esta interfaz.

La propia consulta contiene distintos elementos que definen el origen de datos a utilizar, los datos a recuperar y los diversos filtros y procesamientos. Solo los dos primeros son indispensables en cualquier consulta LINQ. Se representan mediante dos cláusulas situadas, respectivamente, al comienzo y al final de la estructura de la consulta. La cláusula `from` permite definir el origen de datos. La cláusula `select` permite definir un objeto que se devolverá con cada registro. Este objeto puede ser un elemento completo del origen de datos o únicamente un subconjunto de los datos.

Nuestra primera consulta recupera la lista de clientes cuya dirección esté ubicada en España.

```
var consulta = from cli in clientes
               where cli.Pais == "Spain"
               select cli;
```

El tipo de retorno de esta consulta es `IEnumerable<Cliente>`, pero es habitual utilizar la interfaz de tipo con las consultas LINQ, en particular con la cláusula `select` para crear objetos de tipo anónimo.

Los tipos anónimos se describen en el capítulo Programación orientada a objetos con C#, dentro de la sección Tipos anónimos.

Llegados a este punto, a pesar de las apariencias, la variable `consulta` no contiene ningún objeto del tipo `Cliente`. Contiene simplemente una definición de la consulta que se ejecutará bajo demanda, es decir, únicamente cuando sea preciso leer sus datos de retorno. Este mecanismo permite, en particular, ejecutar varias veces la misma consulta LINQ sin tener que redefinirla.

Aquí, la ejecución de la operación de búsqueda se produce mediante el uso de un bucle `foreach`, pero también puede lanzarse para calcular un valor agregado (`Count` o `Max`, por ejemplo) o utilizar ciertos métodos que recuperan varios elementos de alguna manera específica (`First` y `ToList`, por ejemplo).

```
//Muestra el nombre y la dirección de cada cliente
//Muestra también el país para comprobar el resultado
foreach (var resultado in query)
{
    Console.WriteLine(resultado.Nombre);
    Console.WriteLine("\t" + resultado.Direccion);
    Console.WriteLine("\t" + resultado.Ciudad);
    Console.WriteLine("\t" + resultado.Pais);
}
```

El resultado de la ejecución de este código se presenta a continuación. Comprobamos cómo cada uno de los clientes está ubicado en España.

```
Martín Sommer
        C/ Araquil, 67
        Madrid
        Spain
Diego Roel
        C/ Moralzarzal, 86
        Madrid
        Spain
Eduardo Saavedra
        Rambla de Cataluña, 23
        Barcelona
        Spain
José Pedro Freyre
        C/ Romero, 33
        Sevilla
        Spain
Alejandra Camino
        Gran Vía, 1
        Madrid
        Spain
```

2. Los operadores de consulta

LINQ provee numerosos operadores que es importante conocer bien para escribir consultas eficaces. Estos operadores cubren un amplio espectro funcional y pueden agruparse en seis categorías:

- Proyección

- Filtrado
- Ordenación
- Particionado
- Unión y agrupación
- Agregación

Cada operador existe siempre como un método que puede aplicarse a los objetos de tipo `IEnumerable<T>`. Algunos operadores están también integrados en el lenguaje C# y tienen, por tanto, varias palabras clave asociadas. Cuando sea posible, se detallarán los dos modos de escritura.

Los métodos LINQ se utilizan, generalmente, con expresiones lambda. Pueden también utilizarse con funciones y delegados, pero por motivos de legibilidad y de escritura de código "estándar" aquí utilizaremos expresiones lambda.

a. Proyección

Una proyección es un proceso de transformación de un objeto bajo alguna otra forma que consiste, a menudo, en un subconjunto de propiedades del objeto inicial. El tipo de destino no debe declararse necesariamente: como cualquier otro objeto instanciado en C#, puede ser un tipo anónimo.

LINQ proporciona dos implementaciones para la proyección de datos. Estas se encuentran en los métodos `Select` y `SelectMany`.

Select<TResult> (alias C#: select)

El método `Select` transforma un elemento en otro nuevo elemento mediante el uso de una función que asocia un objeto que se pasa como entrada con un objeto que se devuelve.

Los nombres y apellidos de los distintos comerciales se pueden recuperar proporcionando una expresión que crea un objeto de tipo anónimo para cada uno de los objetos de la colección.

```
Console.WriteLine("Nombres de los comerciales:");

var nombresComerciales = comerciales.Select(c => new { Nombre =
c.Nombre, Apellidos = c.Apellidos });
foreach (var com in nombresComerciales)
{
    Console.WriteLine("{0} {1}", com.Nombre, com.Apellidos);
}
```

Es posible reescribir la consulta para utilizar la extensión LINQ del lenguaje C#. La palabra clave asociada es `select`.

```
var nombresComerciales2 = from c in comerciales
                          select new { Nombre = c.Nombre, Apellidos =
c.Apellidos };
```

En cualquier caso, el resultado de la ejecución es el siguiente.

```
Nombres de los comerciales:
Nancy Davolio
Andrew Fuller
Janet Leverling
Margaret Peacock
Steven Buchanan
Michael Suyama
Robert King
Laura Callahan
Anne Dodsworth
```

SelectMany<TResult>

En algunos casos es necesario que la proyección devuelva varios objetos que se integrarán en la colección devuelta por la ejecución de la consulta. Para ello, el método `Select` no está bien adaptado, pues devuelve una colección que contiene varias colecciones:

Para evitar este problema, conviene utilizar el método `SelectMany`. Este método efectúa el mismo procesamiento que el método `Select` pero la expresión que se le pasa debe devolver una colección.

Cabe destacar que la función que se pasa al método `SelectMany` puede devolver una cadena de caracteres, pues se considera como una colección de objetos de tipo `char`. Preste atención, ¡este comportamiento es poco deseable!

b. Filtrado

El filtrado es una funcionalidad esencial en el procesamiento de datos. Es habitual, en efecto, querer limitar el número de registros devueltos proporcionando uno o varios criterios para ello. En la escritura de nuestra primera consulta LINQ, este operador se ha utilizado de manera integrada para devolver únicamente los clientes españoles.

```
var clienteEspaña = from cli in clientes
             where cli.Pais == "Spain"
             select cli;
```

La comparación de la propiedad `Pais` y de la cadena "`Spain`" se realiza mediante operadores estándar de C#. Pueden utilizarse, en efecto, en una consulta LINQ. No obstante, aquí la comparación no resulta óptima. Puede ocurrir, en efecto, que la propiedad contenga el valor "`spain`", y no "`Spain`". Para C# estos dos valores son diferentes, lo cual puede suponer que el retorno de una colección parezca incompleto. En este caso, conviene utilizar el método `Equals` de la clase `string`. Recibe, en efecto, un parámetro que permite hacer que la comparación no tenga en cuenta las mayúsculas y minúsculas.

```
var clientesEspaña = from cli in clientes
                     where cli.Pais.Equals("Spain",
StringComparison.OrdinalIgnoreCase)
                     select cli;
```

El método `Where` necesita que se le pase como parámetro una expresión booleana y devolver una colección cuyos elementos sean del mismo tipo que los de la colección de origen.

```
var clientesEspaña = clientes.Where(cli =>
cli.Pais.Equals("Spain",  StringComparison.OrdinalIgnoreCase));
```

Cuando es necesario cumplir varias condiciones, es posible combinarlas manualmente para formar una expresión compleja.

```
var clientesMadrid = from cli in clientes
                     where cli.Pais.Equals("Spain",
StringComparison.OrdinalIgnoreCase)
                     && cli.Ciudad.Equals("Madrid",
StringComparison.OrdinalIgnoreCase)
                     select cli;
```

Es posible, también, crear varias expresiones `where`, lo cual puede simplificar la lectura de la consulta.

```
var clientesMadrid = from cli in clientes
                     where cli.Pais.Equals("Spain",
StringComparison.OrdinalIgnoreCase)
                     where cli.Ciudad.Equals("Madrid",
StringComparison.OrdinalIgnoreCase)
                     select cli;
```

c. Ordenación

Es muy sencillo recuperar una lista de elementos ordenados en función de uno o varios criterios.

Tras la escritura de una consulta con sintaxis integrada en C#, basta con agregar una cláusula `orderby` seguida de una o varias propiedades separadas por comas. La ordenación se realiza sobre la primera propiedad y, a continuación, si varios valores son idénticos, sobre la segunda propiedad y así sucesivamente.

```
//La cláusula where permite limitar los resultados
var clientesOrdenados = from c in clientes
                        where c.ProductosPedidos.Count < 10
                        orderby c.ProductosPedidos.Count
                        select c;

foreach (var resultado in clientesOrdenados)
{
```

```
    Console.WriteLine(resultado.ProductosPedidos.Count +": " +
resultado.Nombre);
}
```

La ejecución de este código produce el siguiente resultado:

```
0: Diego Roel
0: Marie Bertrand
2: Francisco Chang
2: John Steel
4: Manuel Pereira
6: Martín Sommer
6: Carine  Schmitt
6: Simon  Crowther
6: Dominique Perrier
7: Elizabeth Brown
7: Liz Nixon
8: Eduardo Saavedra
8: Yoshi Tannamuri
8: Liu Wong
9: Janine Labrune
9: Yoshi Latimer
9: Helvetius Nagy
```

Es posible realizar la ordenación en sentido ascendente o descendente agregando las palabras clave `ascending` o `descending` tras el nombre del campo sobre el que se desea especificar el criterio de ordenación. Por defecto, la ordenación es ascendente sobre todos los campos.

El siguiente código agrega una restricción de ordenación descendente sobre la propiedad `Nombre`.

```
//La cláusula where permite limitar los resultados
var clientesOrdenados = from c in clientes
                        where c.ProductosPedidos.Count < 10
                        orderby c.ProductosPedidos.Count, c.Nombre
descending
                        select c;
```

Las palabras clave `orderby`, `ascending` y `descending` se corresponden con cuatro métodos que proporciona LINQ:

- `OrderBy`
- `OrderByDescending`
- `ThenBy`
- `ThenByDescending`

Los dos primeros métodos permiten especificar el primer criterio de ordenación así como el tipo de ordenación asociada. `ThenBy` y `ThenByDescending` tienen el mismo rol pero se aplican para los criterios de ordenación siguientes. Las llamadas a estos métodos pueden encadenarse. La última consulta LINQ escrita utilizando estos

métodos tendría el aspecto siguiente:

```
//El método where permite limitar los resultados
var clientesOrdenados =
    clientes.Where(c => c.ProductosPedidos.Count < 10)
            .OrderBy(c => c.ProductosPedidos.Count)
            .ThenByDescending(c => c.Nombre);
```

d. Particionado

El particionado se corresponde con desacoplar en dos partes un conjunto de datos con el objeto de devolver un único subconjunto. El límite entre ellos puede definirse de manera absoluta o mediante una condición.

El primer caso se implementa mediante el uso de los métodos `Skip` y `Take`. `Skip` permite escindir una colección en base a cierto número de elementos y devolver únicamente la segunda parte. `Take` realiza la tarea inversa y devuelve únicamente el número correspondiente de elementos situados a la cabeza de la colección.

El siguiente código crea una colección que contiene los diez primeros clientes y otra colección que contiene todos los demás elementos.

```
var primerosClientes = clientes.Take(10);
var demasClientes = clientes.Skip(10);
```

Estos métodos se utilizan a menudo en la construcción de sistemas de paginación. Se aplican sobre conjuntos que se han ordenado previamente mediante el método `OrderBy`, lo que permite asegurar que cada elemento aparece en una única página y no se mostrará varias veces, o bien que no se omitirá.

El particionado condicional se aplica mediante el uso de los métodos `SkipWhile` y `TakeWhile`. Estos métodos reciben como parámetro una función cuyo valor de retorno es de tipo booleano. La primera devuelve los elementos de una colección a partir de aquellos que no respetan la condición. El segundo devuelve los elementos en la cabeza de la colección mientras la función que se pasa como parámetro devuelva `true`.

El siguiente ejemplo crea una variable `clientesMasDe30Productos` que contiene la lista de clientes que han pedido más de treinta productos.

```
var clientesMasDe30Productos = clientes.OrderBy(c =>
c.ProductosPedidos.Count()).SkipWhile(c =>
c.ProductosPedidos.Count() <= 30);
```

e. Unión y agrupación

La unión de dos orígenes de datos se corresponde con la asociación automática de estos orígenes mediante una propiedad común a los objetos de ambas colecciones. Sin utilizar uniones sería necesario recorrer manualmente una de las dos fuentes de datos para buscar un elemento correspondiente a cada uno de los objetos de la otra colección.

Con LINQ, esta operación se realiza mediante el operador `join`. Las colecciones y los campos correspondientes se indican a continuación de esta palabra clave.

La siguiente consulta muestra cómo utilizar este operador para recuperar y mostrar el nombre y los apellidos de cada comercial así como la información relativa a los productos solicitados que tiene asociados.

```
var productosYComerciales =
    from c in comerciales
    join c in productosPedidos on c.Identificador equals
p.IdentificadorComercial
    select new
    {
        NombreProducto = p.NombreProducto,
        Cantidad = p.Cantidad,
        PrecioUnitario = p.PrecioUnitario,
        Comercial = string.Format("{0} {1}", c.Nombre, c.Apellidos)
    };

foreach (var productoComercial in productosYComerciales)
{
    Console.WriteLine(productoComercial.NombreProducto);
    Console.WriteLine("\tPrecioUnitario : {0}",
productoComercial.PrecioUnitario);
    Console.WriteLine("\tCantidad : {0}",
productoComercial.Cantidad);
    Console.WriteLine("\tComercial : {0}",
productoComercial.Comercial);
}
```

El método `Join` asocia a esta palabra clave cuatro parámetros:

- La colección implicada en la unión.
- Una función que devuelve el valor que debe utilizarse para unirse con la colección en curso.
- Una función que devuelve el valor que debe utilizarse para unirse con la segunda colección.
- Una función que devuelve los objetos que forman el resultado de la unión.

Las funciones se pasan, generalmente, como expresiones lambda, más simples de leer y más rápidas de escribir.

La consulta anterior podría escribirse con el método `Join` de la siguiente manera:

```
var productosYComerciales =
   comerciales.Join(
       productosPedidos,
       c => c.Identificador,
       p => p.IdentificadorComercial,
       (p, c) => new
       {
           NombreProducto = p.NombreProducto,
           Cantidad = p.Cantidad,
           PrecioUnitario = p.PrecioUnitario,
           Comercial = string.Format("{0} {1}", c.Nombre, c.Apellidos)
       });
```

La propiedad `Comercial` se define para cada uno de los objetos devueltos por la consulta y varios registros pueden tener el mismo valor para esta propiedad. Una manera de hacer que este código sea más eficaz consiste en agrupar los registros por comercial para no tener valores duplicados. La palabra clave `into` definida tras una unión y asociada a un identificador permite, precisamente, realizar esta agrupación.

```
var productosYComercialesGroups =
    from c in comerciales
    join p in productosPedidos on c.Identificador equals
p.IdentificadorComercial into prod
    select new
    {
        Productos = prod,
        Comercial = string.Format("{0} {1}", c.Nombre, c.Apellidos)
    };

foreach (var productoComercial in productosYComercialesGroups)
{
    foreach (var producto in productoComercial.Productos)
    {
        Console.WriteLine(producto.NombreProducto);
        Console.WriteLine("\tPrecioUnitario : {0}",
producto.PrecioUnitario);
        Console.WriteLine("\tCantidad : {0}", producto.Cantidad);
        Console.WriteLine("\tComercial : {0}",
productoComercial.Comercial)
    }
}
```

El método asociado a las palabras clave `join ... into ... no` es `Join` sino `GroupJoin`. Recibe prácticamente los mismos parámetros que el método `Join`, la diferencia entre ambos reside en la función que genera los objetos que forman el resultado de la unión: el segundo parámetro de esta función ya no es de tipo `ProductoPedido` sino de tipo `IEnumerable<ProductoPedido>`.

```
var productosYComerciales4 =
   comerciales.GroupJoin(
      productosPedidos,
      c => c.Identificador,
      p => p.IdentificadorComercial,
      (c, prod) => new
      {
        Productos = prod,
        Comercial = string.Format("{0} {1}", c.Nombre, c.Apellidos)
      }
   );
```

f. Agregación

Los operadores de agregación se definen en LINQ mediante un juego de métodos que devuelven un valor único, generalmente de tipo numérico (aunque no obligatoriamente). El uso de estos métodos es apropiado para:

- Calcular un número de elementos,
- Calcular una media,
- Devolver un valor mínimo o máximo,
- Calcular una suma.

Cálculo del número de elementos: Count

El método `Count` posee dos definiciones. La primera, que no recibe ningún parámetro, devuelve los elementos contenidos en la colección sobre la que se aplica.

```
int numeroElementos = comerciales.Count();
```

La segunda definición recibe un parámetro que permite informar un predicado. El valor devuelto por el método se corresponde con el número de elementos que respetan el predicado.

```
int numeroProductosEconomicos = productosPedidos.Count(p =>
p.PrecioUnitario < 10);
```

Cálculo de una media: Average

El método `Average` recibe como parámetro una función que devuelve un valor numérico para cada objeto de la colección. Realiza sobre estos valores un cálculo de la media y devuelve el resultado.

```
double importeMedioPedido = productosPedidos.Average(p =>
p.PrecioUnitario * p.Cantidad);
```

Búsqueda del valor mínimo: Min

La función `Min` tiene dos definiciones que permiten buscar el valor mínimo en:

- Una colección de objetos numéricos o de tipo,
- Un conjunto de valores numéricos (o de tipo cadena) devueltos por la función que se pasa como parámetro.

```
double precioMenosElevado = productoPedidos.Min(p =>
p.PrecioUnitario);
```

Búsqueda del valor máximo: Max

El método `Max` se utiliza exactamente de la misma manera que `Min`, pero devuelve el valor máximo para un conjunto de elementos.

```
double precioMasElevado = productoPedidos.Max(p =>
p.PrecioUnitario);
```

Cálculo de una suma: Sum

El método `Sum` calcula la suma de los valores devueltos por la función que se le pasa como parámetro.

```
double totalPedidos = productoPedidos.Sum(p => p.PrecioUnitario
* p.Cantidad);
```

Entity Framework

Hemos visto cómo LINQ sabe manipular a la perfección los objetos y sus propiedades y que resulta una herramienta ideal para la manipulación de datos. No obstante, estas colecciones de datos tienen un inconveniente importante: cuando termina la ejecución de la aplicación, se pierden por completo. La solución más común para conservar estos datos de manera persistente consiste en utilizar una base de datos a la que se confía la tarea de salvaguardar de la información.

El objetivo de Entity Framework es realizar de vínculo entre los mundos del desarrollo .NET y las bases de datos con el concepto **mapeo objeto-relacional**.

1. El mapeo objeto-relacional

El modelo tradicional de manipulación de datos, tal como existe con ADO.NET, puede ser difícil de utilizar cuando aumenta el número de tablas o de columnas. Las lecturas y escrituras de datos requieren, en particular, tener cierto rigor en cuanto a la gestión de nombres y tipos de elementos. Para remediar este problema, ha aparecido una técnica de desarrollo de software para manipular los datos según el paradigma de la programación orientada a objetos: el mapeo objeto-relacional (a menudo abreviado como ORM: *Object-Relational Mapping*). Esta técnica proporciona un medio de actuar sobre los orígenes de los datos sin que sea necesario hacer juegos malabares entre el lenguaje de programación y el lenguaje de consultas, con toda la lógica de transformación que necesita el paso de datos de un mundo al otro.

Concretamente, el papel de Entity Framework es permitir la traducción de llamadas a métodos (sobre todo los métodos de la API LINQ) que operan con colecciones de objetos .NET en consultas SQL que la base de datos pueda interpretar y ejecutar. Cuando se devuelven los datos, se convierten automáticamente en objetos directamente utilizables en el código de la aplicación. Estas operaciones implican que se creen tipos .NET para representar la información guardada a nivel del origen de los datos. También es necesario que el motor de mapeo objeto-relacional disponga de información que le permita vincular los tipos de objetos y las tablas de la base de datos. De este modo, las acciones asociadas al origen de los datos se encapsulan en un objeto, simplificando los procesos y reduciendo los costes asociados al desarrollo de una solución lógica con uno o varios orígenes de datos.

Entity Framework permite utilizar tres enfoques diferentes para implementar esta técnica en una aplicación. Cada una tiene sus especificidades sobre el modelo de objetos .NET y los metadatos necesarios para relacionarlas con la base de datos.

El equipo responsable del desarrollo de Entity Framework inicialmente se propuso generar los tipos que representan los datos de la base y almacenar los metadatos utilizados para vincular los mundos .NET y Datos en un archivo externo en formato XML. Se asocian dos enfoques a este modo de funcionamiento: Database First y Model First. A partir de la versión 4.1 de Entity Framework se ha agregado un tercer enfoque, Code First. Este último proporciona una experiencia que se centra sobre todo en el código.

a. Database First

Actualmente se realizan numerosos desarrollos para sustituir aplicaciones antiguas o proporcionar servicios adicionales. Las nuevas aplicaciones se asocian entonces a los datos que las preceden. Database First es un enfoque perfecto para este tipo de escenario. Da la posibilidad a los desarrolladores de conectar su código nuevo con una base de datos existente mediante la generación de tipos .NET y de la información de mapeo necesaria para la manipulación de los datos.

b. Model First

El enfoque Model First está, como Database First, dirigido a la generación de los elementos que constituyen la capa de acceso a los datos: las clases .NET y la información de mapeo. De todos modos, los elementos que

permiten llevar a cabo esta operación no se extraen de la base de datos, sino que se definen en el proyecto mediante un diseñador gráfico integrado en el entorno de desarrollo. El desarrollador se encarga del diseño de un modelo puramente orientado a objetos, que el motor de Entity Framework utilizará como origen para construir una base de datos.

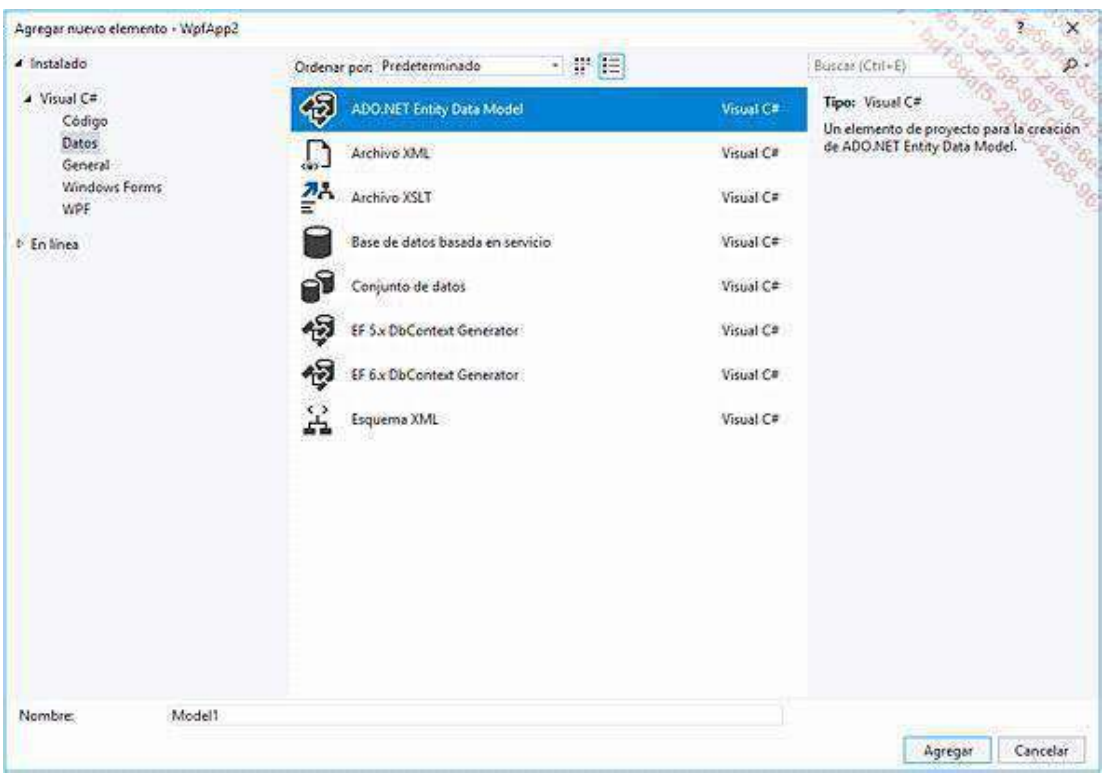
c. Code First

Este último enfoque integrado en Entity Framework puede parecer más natural para los desarrolladores ya que está totalmente orientado al código .NET. El modelo no es generado por el diseñador gráfico, sino que el desarrollador tiene el control. Por el contrario, su nombre puede llevar a pensar que la fase de escritura del código debe ser anterior a la existencia de la base de datos. En realidad, es posible generar el modelo de objetos, como con el enfoque Database First, pero también se puede modificar sin temor ya que no se puede sobreescribir después de una modificación en el diseñador gráfico.

2. Utilización del diseñador objeto/relacional

Una plantilla Entity Framework está formada por entidades y vínculos que las relacionan. El diseñador objeto/relacional de Entity Framework proporciona una representación visual de estos elementos de una forma parecida a la de un modelo UML o la de un modelo conceptual de los datos (método MERISE).

Antes de editar el esquema y el modelo subyacente, evidentemente hay que crearlo. Para ello, Visual Studio proporciona una plantilla que puede seleccionar en el cuadro de diálogo **Agregar nuevo elemento**: **ADO.NET Entity Data Model**.

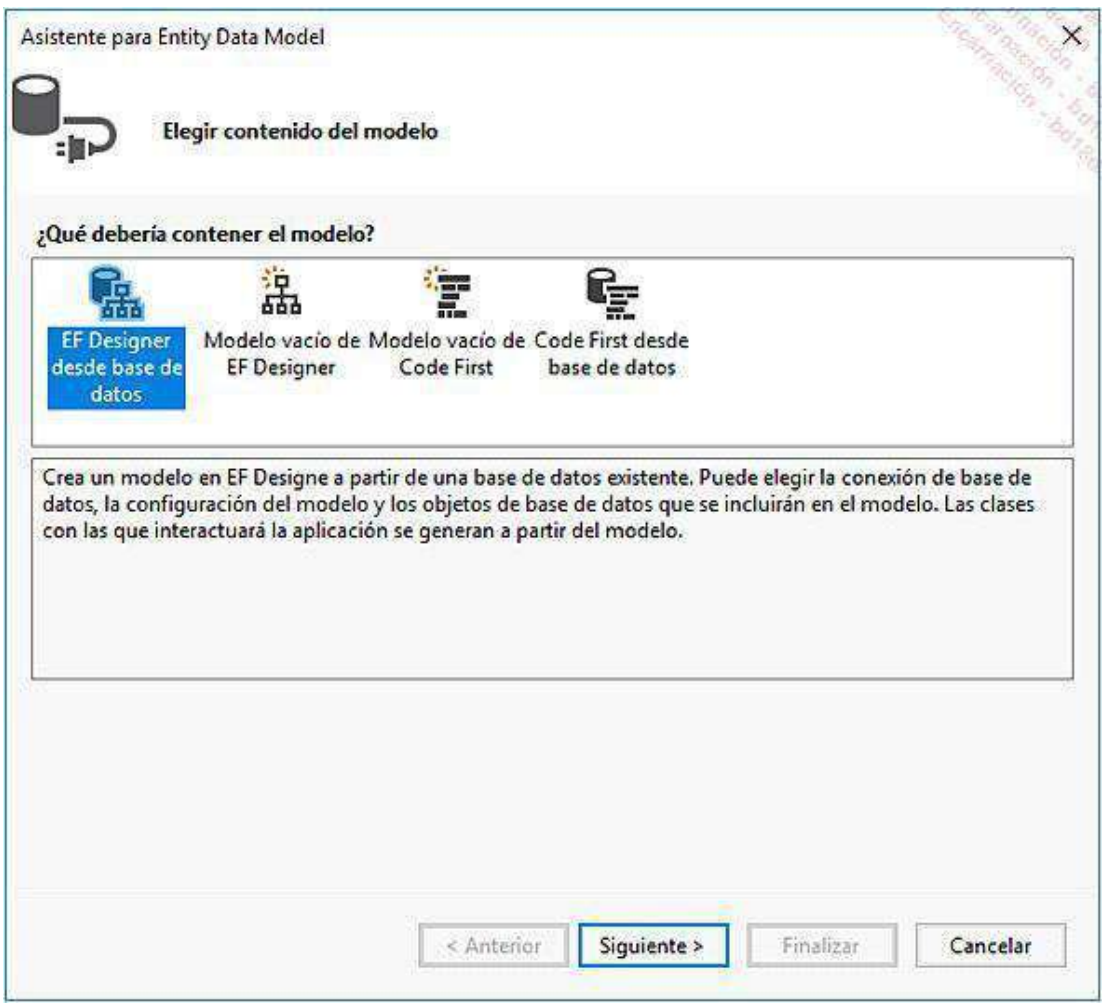


Durante la creación del modelo Entity Framework, se abre un asistente de configuración, el **asistente EDM**. Permite definir el enfoque elegido para la creación del modelo de objetos a través de cuatro opciones:

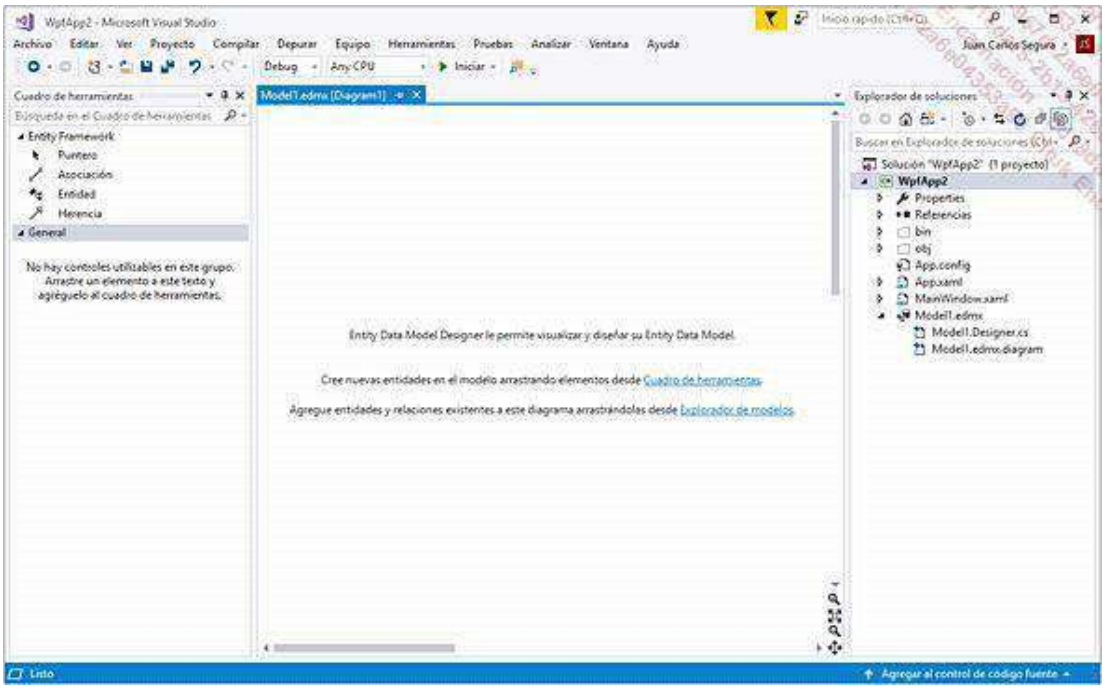
- **EF Designer desde base de datos**: esta opción corresponde al enfoque Database First. La información relativa a la estructura de la base de datos se recupera y se guarda de modo que el motor de Entity Framework pueda generar el

modelo de objetos .NET.

- **Modelo vacío de EF Designer**: representa el enfoque Model First. Cada modificación realizada en el diseñador comporta la actualización de los metadatos guardados localmente así como la regeneración de los tipos .NET que componen el modelo de objetos.
- **Modelo vacío de Code First**: como su nombre indica, está asociado al enfoque Code First. Por este motivo, no genera ningún archivo editable con el diseñador gráfico pero sí un tipo .NET que hereda de `DbContext`. Este será el punto de partida de un modelo de objetos completo.
- **Code First desde base de datos**: como la plantilla anterior, genera un archivo de código y no un archivo editable en el diseñador gráfico. Por el contrario, "rellena" el modelo de objetos con información del origen de datos.



Si se crea un modelo vacío, inmediatamente se abre el diseñador gráfico sin ningún elemento.



En el proyecto aparecen varios archivos: un archivo .edmx, un archivo .Designer.cs y un archivo .edmx.diagram.

El archivo .edmx

Este archivo almacena todos los metadatos necesarios para la generación de un modelo de objetos. Cuando se abre con un doble clic, su contenido se utiliza para mostrar un esquema que representa el modelo en el diseñador gráfico. Cuando se abre en modo texto, muestra toda la información guardada en formato XML. A continuacuón puede ver el contenido del archivo asociado a un modelo de objetos vacío:

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="3.0" xmlns:edmx="http://schemas.microsoft.com/ado/
2009/11/edmx">
  <!-- EF Runtime content -->
  <edmx:Runtime>
    <!-- SSDL content -->
    <edmx:StorageModels>
      <Schema xmlns="http://schemas.microsoft.com/ado/2009/11/edm/ssdl"
Namespace="Model1.Store" Alias="Self" Provider="System.Data.SqlClient"
ProviderManifestToken="2005">
        <EntityContainer Name="Model1TargetContainer" >
        </EntityContainer>
      </Schema>
    </edmx:StorageModels>
    <!-- CSDL content -->
    <edmx:ConceptualModels>
      <Schema xmlns="http://schemas.microsoft.com/ado/2009/11/edm"
xmlns:cg="http://schemas.microsoft.com/ado/2006/04/codegeneration"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/
EntityStoreSchemaGenerator" Namespace="Model1" Alias="Self" xmlns:
annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation"
annotation:UseStrongSpatialTypes="false">
        <EntityContainer Name="Model1Container"
```

```
annotation:LazyLoadingEnabled="true">
        </EntityContainer>
      </Schema>
    </edmx:ConceptualModels>
    <!-- C-S mapping content -->
    <edmx:Mappings>
      <Mapping xmlns="http://schemas.microsoft.com/ado/2009/11/
mapping/cs" Space="C-S">
        <Alias Key="Model" Value="Model1" />
        <Alias Key="Target" Value="Model1.Store" />
        <EntityContainerMapping CdmEntityContainer="Model1Container"
StorageEntityContainer="Model1TargetContainer">
        </EntityContainerMapping>
      </Mapping>
    </edmx:Mappings>
  </edmx:Runtime>
  <!-- EF Designer content (DO NOT EDIT MANUALLY BELOW HERE) -->
  <edmx:Designer xmlns="http://schemas.microsoft.com/ado/2009/11/edmx">
    <edmx:Connection>
      <DesignerInfoPropertySet>
        <DesignerProperty Name="MetadataArtifactProcessing" Value=
"EmbedInOutputAssembly" />
      </DesignerInfoPropertySet>
    </edmx:Connection>
    <edmx:Options>
      <DesignerInfoPropertySet>
        <DesignerProperty Name="ValidateOnBuild" Value="true" />
        <DesignerProperty Name="EnablePluralization" Value="False" />
        <DesignerProperty Name="CodeGenerationStrategy" Value="Ninguno" />
      </DesignerInfoPropertySet>
    </edmx:Options>
    <!-- Diagram content (shape and connector positions) -->
    <edmx:Diagrams>
    </edmx:Diagrams>
  </edmx:Designer>
</edmx:Edmx>
```

Este archivo contiene cuatro secciones principales, identificadas en negrita, que describen respectivamente:

- La estructura del origen de los datos: SSDL content
- La estructura del modelo de objetos: CSDL content
- La relación entre estos dos elementos (dicho de otro modo, el mapeo) : C-S mapping content
- Información general relativa al modelo (opciones de generación o datos asociados a la visualización en el diseñador): EF Designer content

El archivo .Designer.cs

Cuando se agrega, modifica o elimina una entidad o un vínculo entre entidades mediante el diseñador gráfico, se desencadena un proceso de generación de código. El resultado de esta operación se guarda en el archivo .Designer.cs asociado al fichero .edmx editado. De todos modos, en el caso del enfoque Model First, no se realiza ninguna generación antes de que la versión de Entity Framework que se debe utilizar no esté definida. Para utilizar el modelo en el código, es esencial gestionar esta problemática añadiendo explícitamente un generador de

código Entity Framework 5 o 6 o creando una base de datos a partir del modelo, lo que agrega implícitamente un generador Entity Framework 6.

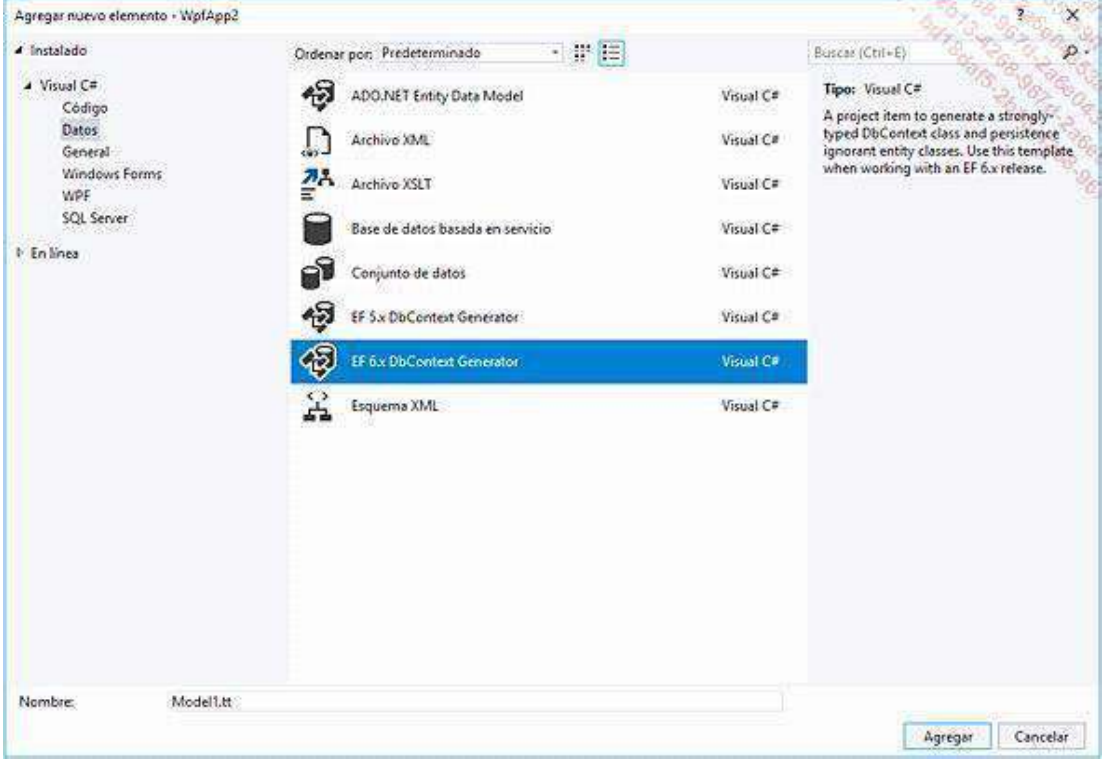
El archivo .edmx.diagram

Con el tiempo, la integración de los datos relativos a la visualización del modelo en el diseñador ha mostrado sus límites: modificaciones menores frecuentes al menor arrastrar-soltar (voluntario o no) y conflictos a nivel de los administradores de código fuente en cada desplazamiento de una entidad, lo que entraña el riesgo de perder información esencial en caso de una mala gestión. Estos datos se han desplazado al archivo .edmx.diagram, que gestiona exclusivamente los elementos relativos a la visualización en el diseñador. Esto ha permitido eliminar los problemas citados anteriormente: basta con sobreescribir sistemáticamente el archivo .edmx.diagram cuando se guarde el código.

Agregar un generador Entity Framework con Model First

Esta operación, que es extremadamente sencilla, es esencial en el marco de un desarrollo ya que es la que permite la generación automática de un modelo de objetos a partir de un esquema conceptual.

El menú contextual muestra cómo agregar un generador con la opción **Agregar un elemento de generación de código**. Al seleccionarla, esta opción abre el cuadro de diálogo para agregar un nuevo archivo e incluye en la lista los diferentes generadores instalados.

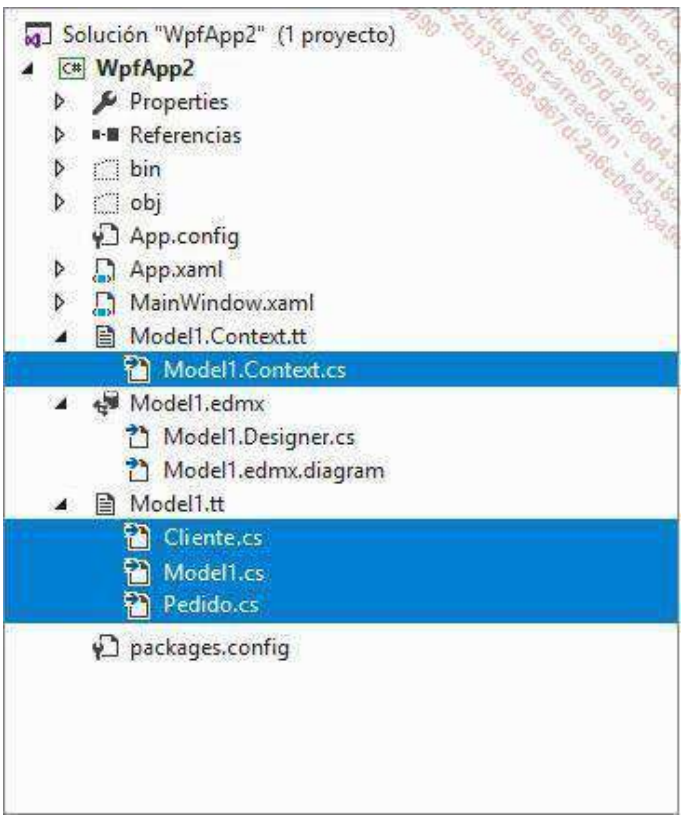


La selección de **EF 6.x DbContext Generator** agrega dos nuevos archivos al proyecto, que tienen como extensión respectivamente .tt y .Context.tt. Son archivos de modelo que utilizan el framework T4 (*Text Template Transformation Toolkit*) de Microsoft. El archivo .tt define el formato de las diferentes entidades cuyo código se ha generado, mientras que el archivo .Context.tt está destinado a generar un tipo contenedor, heredado de `DbContext`, que será el punto de entrada para la manipulación de entidades y la repercusión en base de datos de las modificaciones que se realicen. Se puede comparar este segundo tipo a una base de datos de objetos .NET.

La fase de compilación del proyecto ejecuta ahora, mediante el esquema .edmx, un proceso de generación de

código basado completamente en estos dos archivos de modelo. Para dos entidades, hay cuatro archivos generados:

- Un archivo .Context.cs, con el tipo de contenedor de todas las entidades.
- Dos archivos que contienen el código de las entidades.
- Un archivo .cs (en este caso, Model1.cs) vacío, pero presente por razones de compatibilidad con versiones anteriores.



A continuación se puede ver la totalidad del código generado por un modelo simple que contiene dos entidades vinculadas entre sí.

Archivo Model1.Context.cs

```
namespace WpfApp2
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    public partial class Model1Container1 : DbContext
    {
        public Model1Container1()
            : base("name=Model1Container1")
        {
        }

        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
```

```
        {
            throw new UnintentionalCodeFirstException();
        }

        public virtual DbSet<Cliente> ClienteSet { get; set; }
        public virtual DbSet<Pedido> PedidoSet { get; set;  }
    }
}
```

Archivo Cliente.cs

```
namespace WpfApp2
{
    using System;
    using System.Collections.Generic;

    public partial class Cliente
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2214:DoNotCallOverridableMethodsInConstructors")]
        public Cliente()
        {
            this.Motel = new HashSet<Motel>();
        }

        public int Id { get; set; }
        public string Nombre { get; set; }
        public string Poblacion { get; set; }
        public string Property1 { get; set;  }

        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2227:CollectionPropertiesShouldBeReadOnly")]
        public virtual ICollection<Pedido> Pedido { get; set; }
    }
}
```

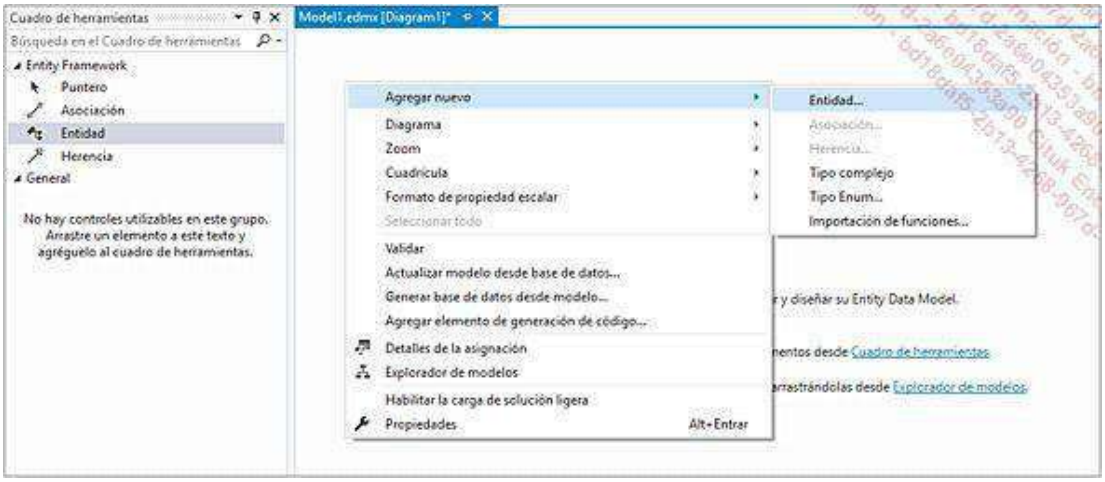
Archivo Pedido.cs

```
namespace WpfApp2
{
    using System;
    using System.Collections.Generic;

    public partial class Pedido
    {
        public int Id { get; set; }
        public string Fecha { get; set;  }
        public string Importe { get; set; }
        public int ClienteId { get; set;  }
```

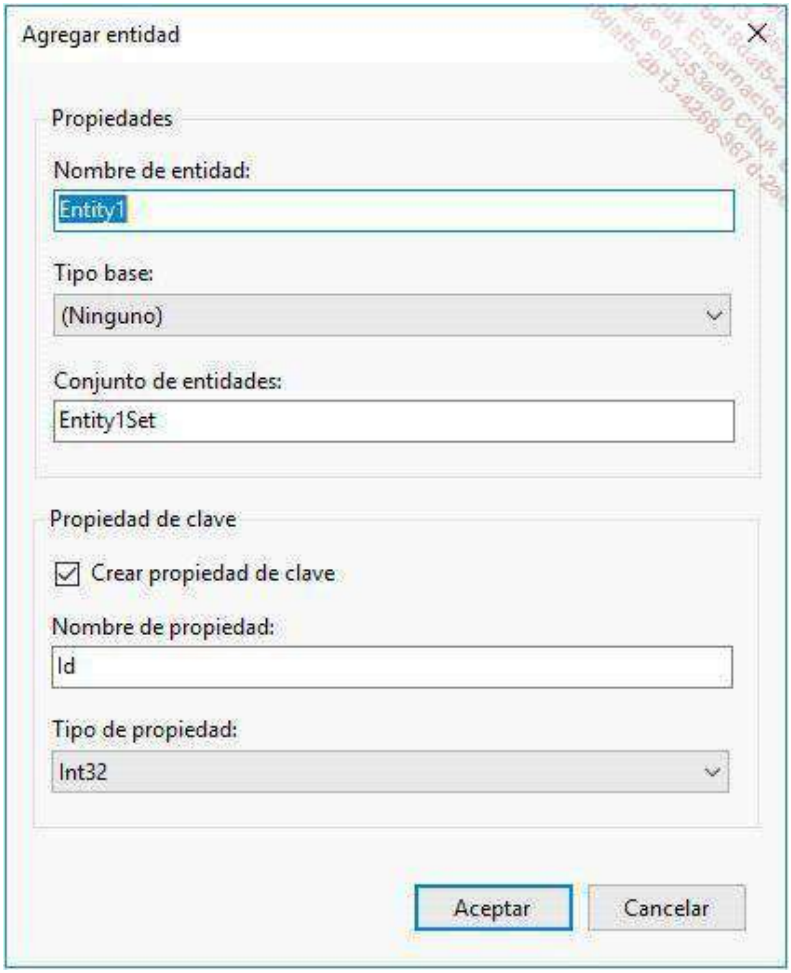
```
        public virtual Cliente Cliente { get; set;  }
    }
}
```

Agregar una entidad

Para agregar una entidad, basta con utilizar la opción **Agregar nuevo - Entidad** en el menú contextual del diseñador o arrastrar y soltar el elemento **Entitad** del **Cuadro de herramientas** en el diseñador gráfico.



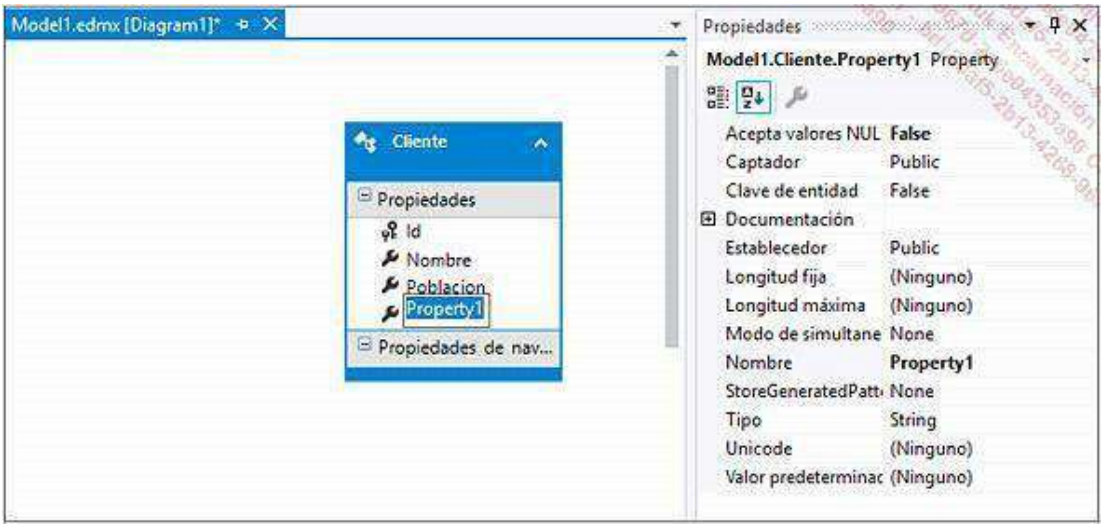
El uso de la primera técnica abre una ventana para configurar la entidad que se va a agregar. Los elementos que se pueden personalizar son sobre todo el nombre de su tipo o de la propiedad que permite acceder a la tabla subyacente desde el contexto de los datos. Este asistente también permite agregar a la entidad una propiedad definida como clave, para que Entity Framework la pueda manipular.



El uso de arrastar y soltar para agregar la entidad es más rápido, ya que no abre ninguna ventana emergente. Estos son los valores por defecto que aparecen:

- **Nombre de entidad**: formato `Entity<numerador>` (p. ej.: Entity1, Entity2...).
- **Tipo base**: ninguno.
- **Conjunto de entidades**: `<Nombre de la entidad>Set`.
- **Crear propiedad de clave**: marcada. La clave se define con el nombre `Id` y el tipo `Int32`.

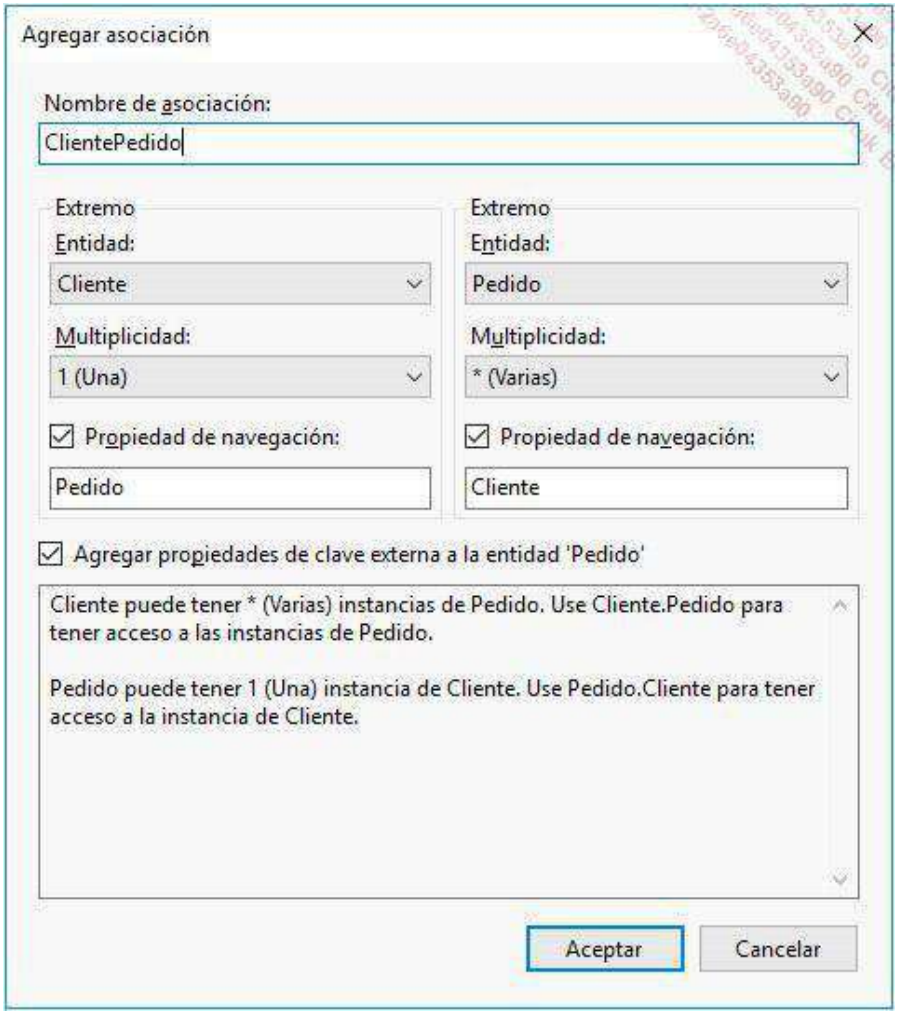
Después de hacer clic en **Aceptar**, la nueva entidad se visualiza en el diseñador. Se pueden editar sus propiedades de manera que se refleje la estructura de base de datos deseada. El uso de la opción **Agregar nueva - Propiedad escalar**del menú contextual de la entidad agrega una propiedad cuyos atributos pueden modificarse desde la ventana **Propiedades**.



Agregar una relación entre entidades

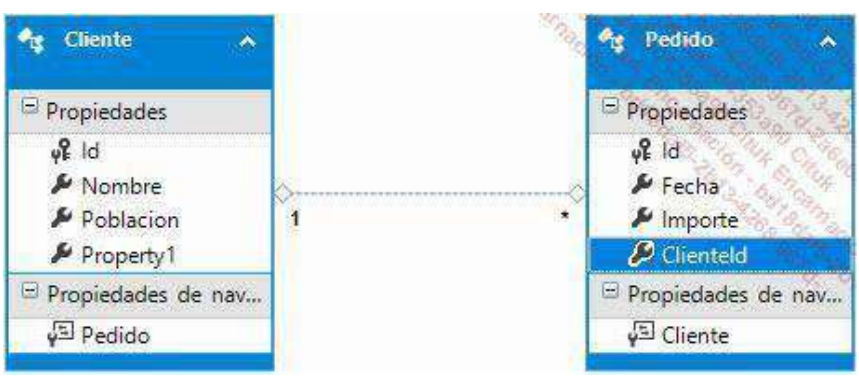
Es posible agregar asociaciones en el diseñador visual cuando se presenta una clase en su superficie de edición. Las asociaciones son el objeto equivalente a las relaciones que pueden existir entre las tablas de la base de datos. Cuando existe una relación entre dos tablas mapeadas, se agrega una asociación automáticamente para reflejar el vínculo entre las tablas.

Es posible crear un vínculo entre entidades con la opción **Agregar nueva - Relación** del menú contextual del diseñador. Igual que para agregar una entidad, el uso de esta técnica abre una ventana de personalización de la relación.



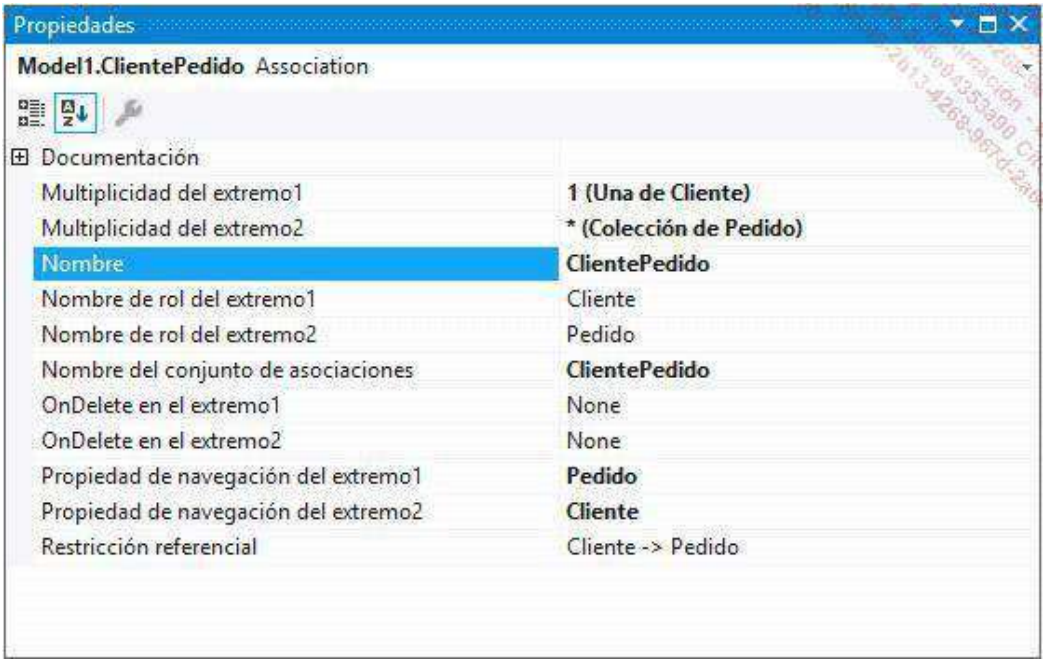
Los diferentes elementos que se muestran en esta interfaz permiten definir los dos extremos de la relación: las entidades implicadas así como la cardinalidad que se asocia a cada una de ellas. Para simplificar la selección y la actualización de los registros de ambas entidades, Entity Framework utiliza el concepto de **propiedad de navegación**. Estas propiedades tienen la particularidad de ser un tipo de entidad y de tomar valor automáticamente en la selección de datos. Por lo tanto, en vez de buscar un cliente y los pedidos realizados por separado, es posible acceder a los pedidos directamente a través de la colección `Pedidos` del objeto `Cliente`. Para ayudar a los desarrolladores, hay una área de texto que contiene una descripción de la configuración que se está editando.

Después de hacer clic en **Aceptar**, el esquema se actualiza para reflejarlo: la relación se materializa con un vínculo entre las entidades, y las propiedades de navegación se agregan en una sección propia de cada entidad.



El elemento **Relación** del **Cuadro de herramientas** también permite llegar al mismo lugar pero sin la ayuda de un

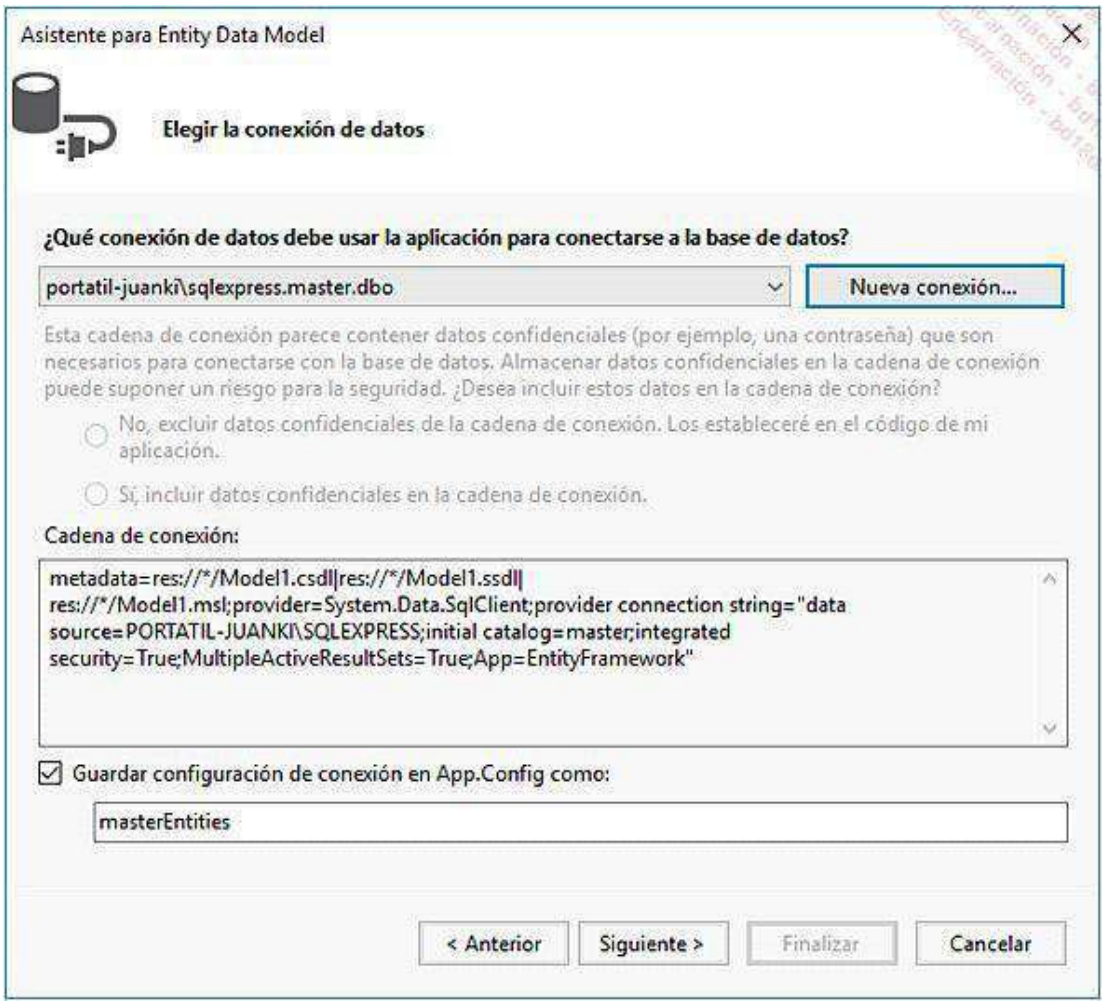
asistente: la configuración inicial tiene valores por defecto y se pueden modificar desde la ventana **Propiedades**.



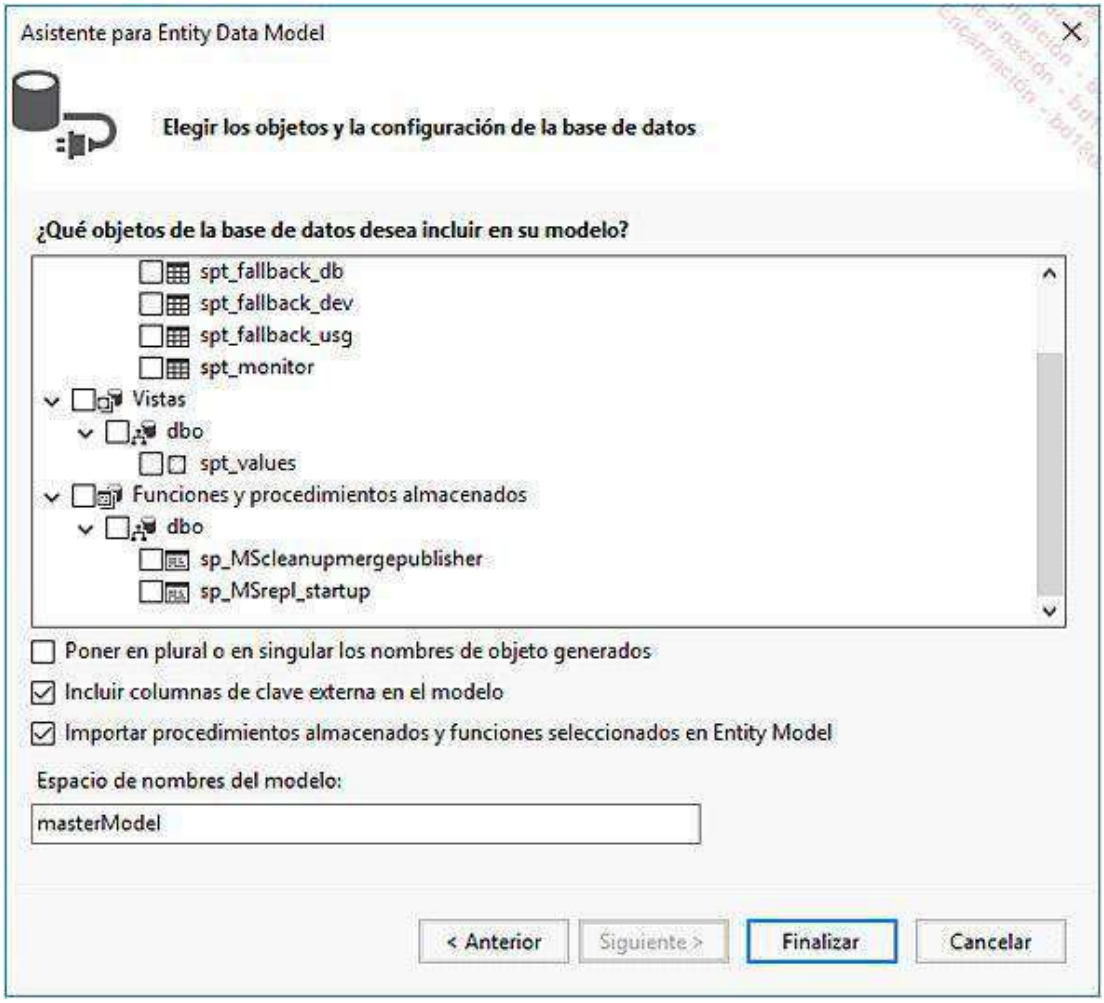
Vincular un modelo y una base de datos

No es necesario vincular un modelo de objetos Entity Framework con una base de datos en la fase de diseño. De todos modos, esta librería permite, una vez se ha asociado el modelo a un origen de datos compatible, reflejar cualquier modificación realizada en el modelo sobre la base de datos y viceversa.

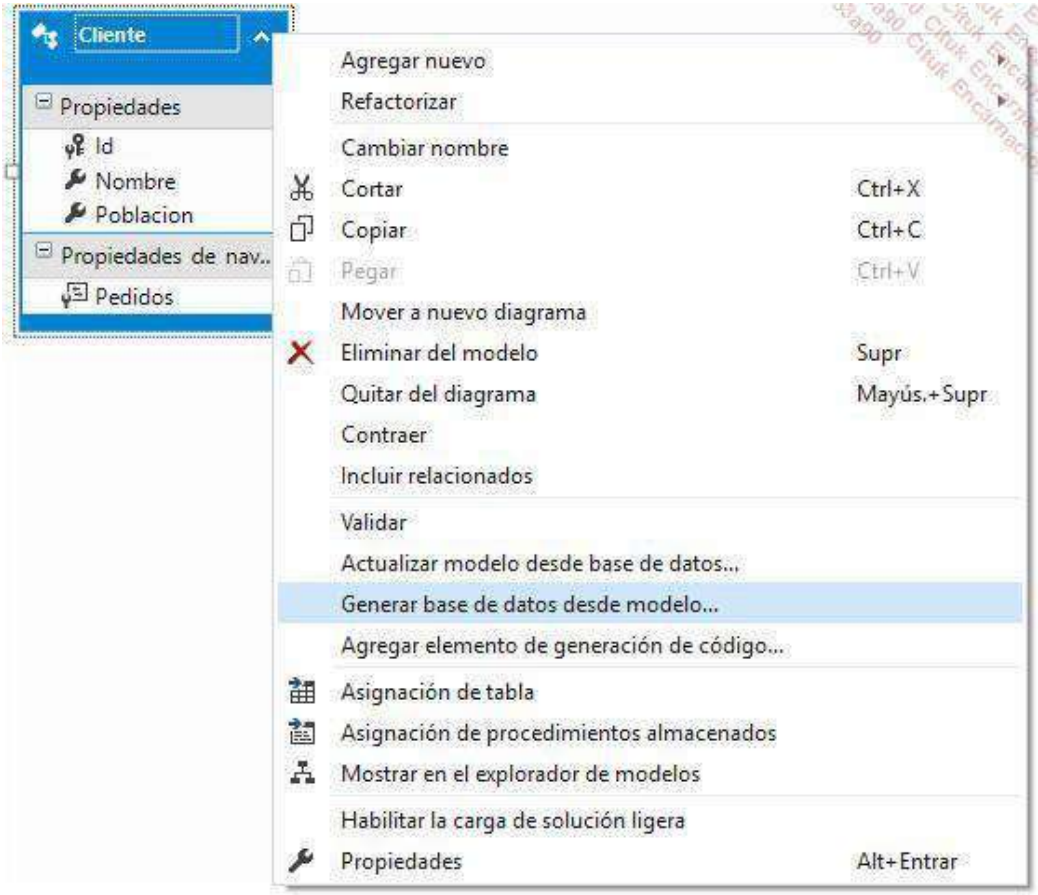
Esta conexión se hace naturalmente cuando se usa el enfoque Database First. Cuando el asistente de creación de un modelo permite elegir el tipo de contenido desde el que se cargarán los datos, basta con elegir **EF Designer desde base de datos**. El asistente mostrará una página de selección de la base de datos de origen.



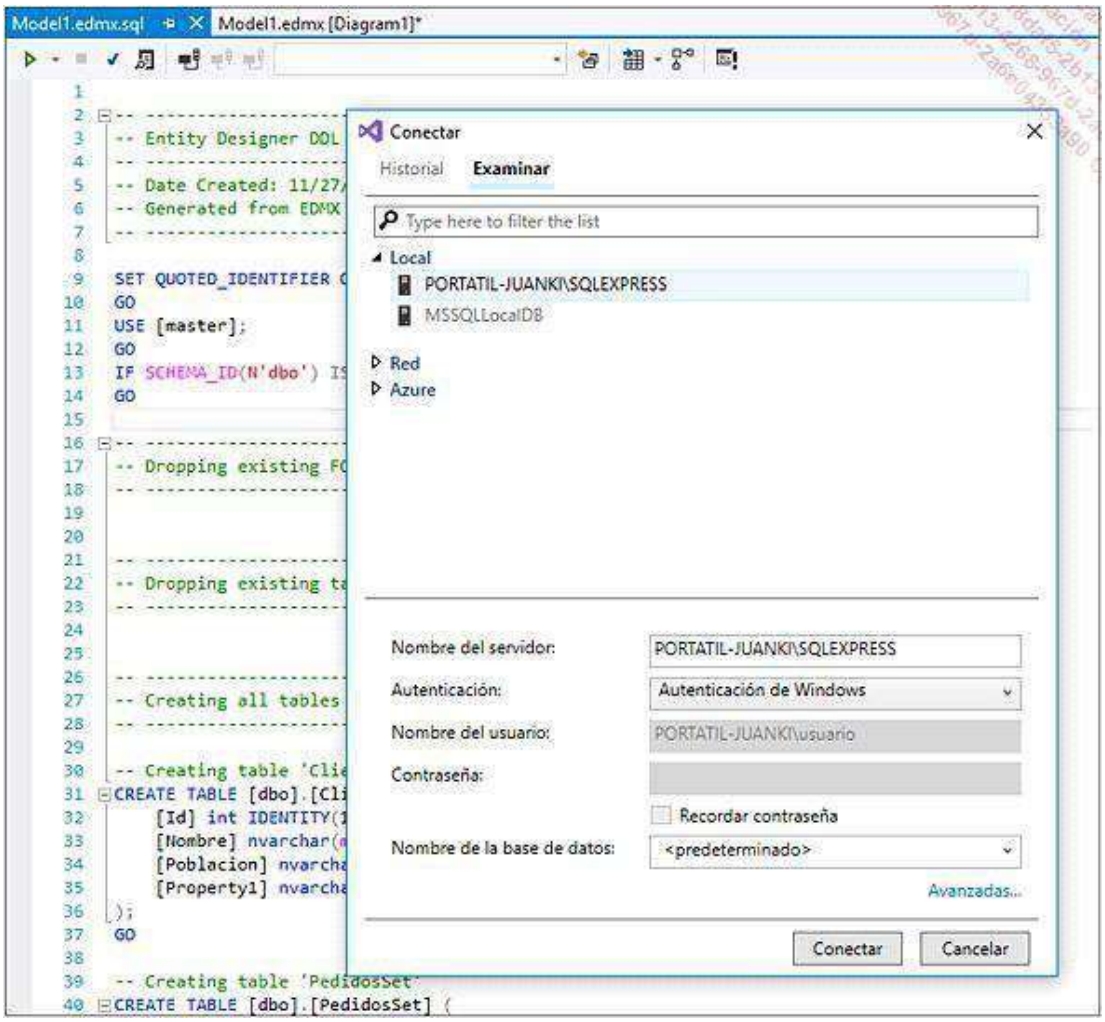
Después de la selección del origen de datos se seleccionan los elementos (tablas, vistas, funciones y procedimientos almacenados) que se deben integrar en el modelo Entity Framework.



Cuando el modelo se ha creado independientemente de la base de datos (Model First), la relación se hace mediante la ejecución explícita de una operación vinculada a la base de datos: generación de la base de datos a partir del modelo o actualización del modelo a partir de un origen de datos. Con Entity Framework, estos dos procesos se desencadenan a partir del menú contextual del diseñador gráfico.



En el primer caso, se abre la interfaz de selección de bases de datos vista anteriormente y a continuación, en una nueva pestaña de Visual Studio, se genera un script SQL que crea la estructura de base de datos. Si se hace clic en el botón **Ejecutar** se muestra una ventana de conexión con el origen de datos.



Cuando se realiza la conexión, se ejecuta el script SQL y se crean en la base de datos las tablas asociadas a las entidades.

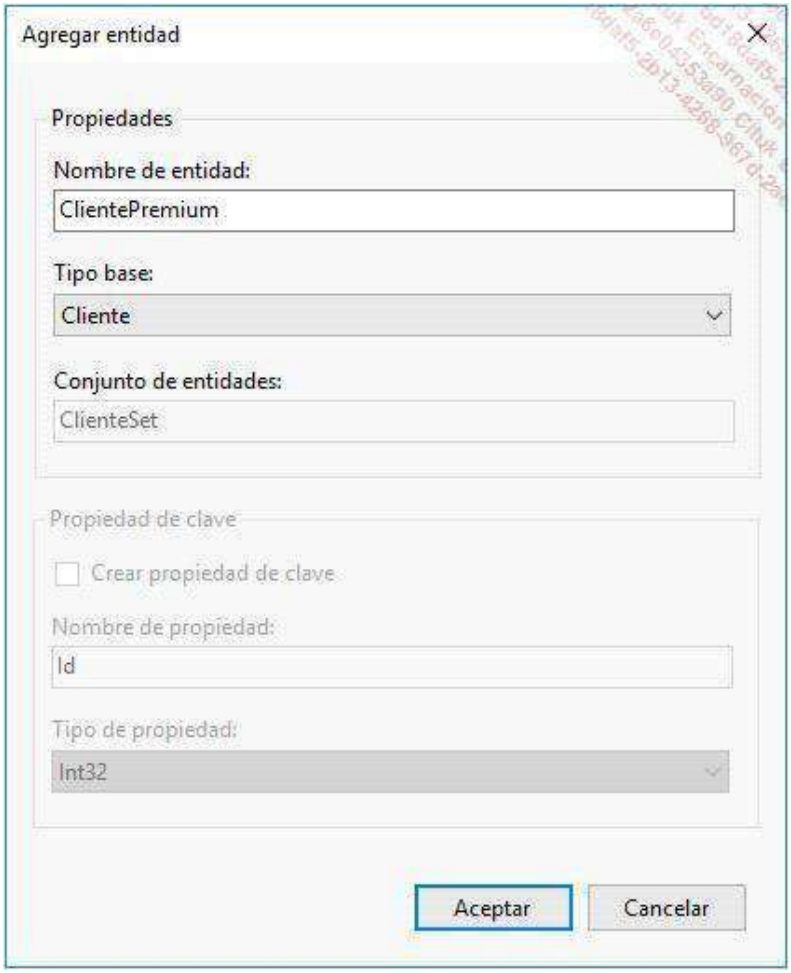
Herencia

El diseñador objeto/relacional soporta la creación de relaciones de herencia. Esta noción está perfectamente integrada en C#, pero no existe en las bases de datos relacionales. Es necesario, por tanto, utilizar una solución de rodeo para simular con éxito la herencia en una base de datos.

La relación de herencia se puede representar como el uso de una tabla como miembro de la relación. Cada tabla contiene la información vinculada a un tipo y los datos se agrupan en la lectura.

En la relación descrita en el esquema anterior, un objeto de tipo `ClientePremium` se guarda en parte en la tabla asociada a la entidad `Cliente`, mientras que el miembro `Estado` se guarda en la tabla asociada a la entidad `ClientePremium`. El único problema real con este enfoque es mantener el vínculo entre las dos partes del registro. Para ello, Entity Framework utiliza la clave única de la entidad `Cliente`: el valor de clave se utiliza en ambas tablas, de manera que se pueden vincular fácilmente.

Para crear la relación, cuando se agrega una entidad hijo, hay que seleccionar el tipo de entidad padre en el cuadro de diálogo de configuración de la entidad. El asistente desactiva entonces automáticamente los elementos relativos a la definición de una clave primaria, de manera que el motor de Entity Framework guarda su valor.



3. Uso de LINQ con Entity Framework

Entity Framework permite utilizar la sintaxis LINQ descrita en la primera sección de este capítulo para interrogar a bases de datos SQL Server. De todos modos, con Entity Framework, los objetos manipulados se crean a partir de los datos propios de la base de datos.

La base de datos utilizada en todos los ejemplos de esta sección es la base de datos Northwind de Microsoft. El script SQL para crearla se puede descargar desde la página Información. El modelo `NorthwindDataContext` se ha generado en Database First.

a. Recuperación de datos

La comunicación con la base de datos se gestiona gracias al `DbContext`, lo que implica que es obligatorio utilizar esta clase para acceder a los datos.

Esta clase expone distintas colecciones que se corresponden con las distintas tablas mapeadas en el diseñador visual. El código que define la propiedad vinculada con la tabla `Customers` en el `DbContext` es el siguiente:

```
public virtual DbSet<Customers> Customers
{
    get ; set ; }
```

La recuperación de los datos de la tabla `Customers` se realiza mediante una consulta LINQ que implica a la colección asociada. La visualización de la lista de clientes españoles se realiza de la siguiente manera:

```
NorthwindDataContext context = new NorthwindDataContext();

var clientesEspañoles = from c in context.Customers
                                    where c.Country == "Spain"
                                    select c;

foreach (var cliente in clientesEspañoles)
{
    Console.WriteLine(cliente.CompanyName);
}
```

La clase padre de `NorthwindDataContext`, `DbContext`, implementa la interfaz `IDisposable`. Conviene, por tanto, utilizar la estructura `using` para asegurar que se liberan los recursos vinculados a los objetos de este tipo.

Es posible mostrar la consulta SQL generada mediante una consulta LINQ asignando valor a la propiedad `Log` del objeto `DbContext.Database` escribiendo en la consola:

```
context.Database.Log = Console.WriteLine;
```

En el caso de nuestra búsqueda de clientes españoles, el código SQL ejecutado es:

```
SELECT
[Extent1].[CustomerID] AS [CustomerID],
[Extent1].[CompanyName] AS [CompanyName],
[Extent1].[ContactName] AS [ContactName],
[Extent1].[ContactTitle] AS [ContactTitle],
[Extent1].[Address] AS [Address],
[Extent1].[City] AS [City],
[Extent1].[Region] AS [Region],
[Extent1].[PostalCode] AS [PostalCode],
[Extent1].[Country] AS [Country],
[Extent1].[Phone] AS [Phone],
[Extent1].[Fax] AS [Fax]
FROM [dbo].[Customers] AS [Extent1]
WHERE N'Spain' =  [Extent1].[Country]
```

La consulta generada se corresponde exactamente con el código LINQ y habría podido ejecutarse fácilmente utilizando ADO.NET. Entity Framework empieza a desvelar su potencia cuando se complican las consultas. El siguiente ejemplo muestra las fechas de los pedidos por cliente.

```
using (NorthwindDataContext context = new NorthwindDataContext())
{
    var pedidosClientes =
        from c in context.Customers
```

```
        join pedido in context.Orders on c.CustomerID equals
pedido.CustomerID into pedidosCliente
        select new { Cliente = c, Pedidos = pedidosCliente };

    foreach (var pedidoCli in pedidosClientes)
    {
        Console.WriteLine(pedidoCli.Cliente.CompanyName);
        foreach (var pedido in pedidoCli.Pedidos)
        {
            Console.WriteLine(pedido.OrderDate);
        }
    }
}
```

El código SQL generado es el siguiente:

```
SELECT
    [Project1].[C1] AS [C1],
    [Project1].[CustomerID] AS [CustomerID],
    [Project1].[CompanyName] AS [CompanyName],
    [Project1].[ContactName] AS [ContactName],
    [Project1].[ContactTitle] AS [ContactTitle],
    [Project1].[Address] AS [Address],
    [Project1].[City] AS [City],
    [Project1].[Region] AS [Region],
    [Project1].[PostalCode] AS [PostalCode],
    [Project1].[Country] AS [Country],
    [Project1].[Phone] AS [Phone],
    [Project1].[Fax] AS [Fax],
    [Project1].[C2] AS [C2],
    [Project1].[OrderID] AS [OrderID],
    [Project1].[CustomerID1] AS [CustomerID1],
    [Project1].[EmployeeID AS [EmployeeID],
    [Project1].[OrderDate] AS [OrderDate],
    [Project1].[RequiredDate] AS [RequiredDate],
    [Project1].[ShippedDate] AS [ShippedDate],
    [Project1].[ShipVia] AS [ShipVia],
    [Project1].[Freight] AS [Freight],
    [Project1].[ShipName] AS [ShipName],
    [Project1].[ShipAddress] AS [ShipAddress],
    [Project1].[ShipCity] AS [ShipCity],
    [Project1].[ShipRegion] AS [ShipRegion],
    [Project1].[ShipPostalCode] AS [ShipPostalCode],
    [Project1].[ShipCountry] AS [ShipCountry]
    FROM ( SELECT
        [Extent1].[CustomerID] AS [CustomerID],
        [Extent1].[CompanyName] AS [CompanyName],
        [Extent1].[ContactName] AS [ContactName],
        [Extent1].[ContactTitle] AS [ContactTitle],
        [Extent1].[Address] AS [Address],
        [Extent1].[City] AS [City],
        [Extent1].[Region] AS [Region],
        [Extent1].[PostalCode] AS [PostalCode],
```

```
        [Extent1].[Country] AS [Country],
        [Extent1].[Phone] AS [Phone],
        [Extent1].[Fax] AS [Fax],
        1 AS [C1],
        [Extent2].[OrderID] AS [OrderID],
        [Extent2].[CustomerID] AS [CustomerID1],
        [Extent2].[EmployeeID] AS [EmployeeID],
        [Extent2].[OrderDate] AS [OrderDate],
        [Extent2].[RequiredDate] AS [RequiredDate],
        [Extent2].[ShippedDate] AS [ShippedDate],
        [Extent2].[ShipVia] AS [ShipVia],
        [Extent2].[Freight] AS [Freight],
        [Extent2].[ShipName] AS [ShipName],
        [Extent2].[ShipAddress] AS [ShipAddress],
        [Extent2].[ShipCity] AS [ShipCity],
        [Extent2].[ShipRegion] AS [ShipRegion],
        [Extent2].[ShipPostalCode] AS [ShipPostalCode],
        [Extent2].[ShipCountry] AS [ShipCountry],
        CASE WHEN ([Extent2].[OrderID] IS NULL) THEN CAST(NULL
AS int) ELSE 1 END AS [C2]
        FROM [dbo].[Customers] AS[Extent1]
        LEFT OUTER JOIN [dbo].[Orders] AS Extent2]
ON [Extent1].[CustomerID] =  [Extent2].[CustomerID]
    ) AS [Project1]
    ORDER BY [Project1].[CustomerID] ASC, [Project1].[C2] ASC
```

b. Actualización de datos

La actualización de los datos contenidos en la base de datos se representa mediante tres tipos de acciones: inserción, modificación y eliminación. Estas operaciones pueden ejecutarse, las tres, mediante el uso de Entity Framework.

Inserción de datos

Es posible agregar un registro creando un objeto cuyas propiedades tengan valor. Una vez completo el objeto, este se agrega a la tabla mediante la colección que tiene asociada en el `DbContext`. Por último, se invoca al método `SaveChanges` del `DbContext` para agregarlo a la base de datos mediante la generación y ejecución de una consulta SQL adaptada.

El siguiente ejemplo crea un nuevo cliente en la base de datos.

```
Customers cliente = new Customers();
cliente.CustomerID = "JMART";
cliente.CompanyName = "Mi empresa";
cliente.Address = "17, calle principal";
cliente.City = "Valencia";
cliente.ContactName = "Juan MARTIN";
cliente.Country = "Spain";
cliente.Phone = "650102034";

using (NorthwindDataContext context = new NorthwindDataContext())
{
```

```
    context.Customers.Add(cliente);
    context.SaveChanges();
}
```

Modificación de datos

Entity Framework permite modificar un registro en la base de datos de una manera perfectamente natural. El registro que se desea modificar se recupera, en primer lugar, como un objeto mediante una consulta LINQ. Las distintas propiedades del objeto pueden modificarse a continuación. La última etapa consiste en invocar, como con la acción de inserción, al método `SaveChanges` del `DbContext`. El siguiente código realiza la modificación del nombre de la empresa para el cliente que acabamos de crear:

```
using (NorthwindDataContext context = new NorthwindDataContext())
{
    var cliente = (from c in context.Customers
                           where c.CustomerID == "JMART"
                           select c).First();

    cliente.CompanyName = "Mi nueva empresa";

    context.SaveChanges();
}
```

Eliminación de datos

La última acción de actualización de datos es relativa a la eliminación de un elemento. Se realiza pasando el objeto correspondiente al registro que se desea eliminar al método `DeleteOnSubmit` de la colección vinculada a la tabla correspondiente.

Es posible eliminar el cliente que acabamos de crear de la siguiente manera:

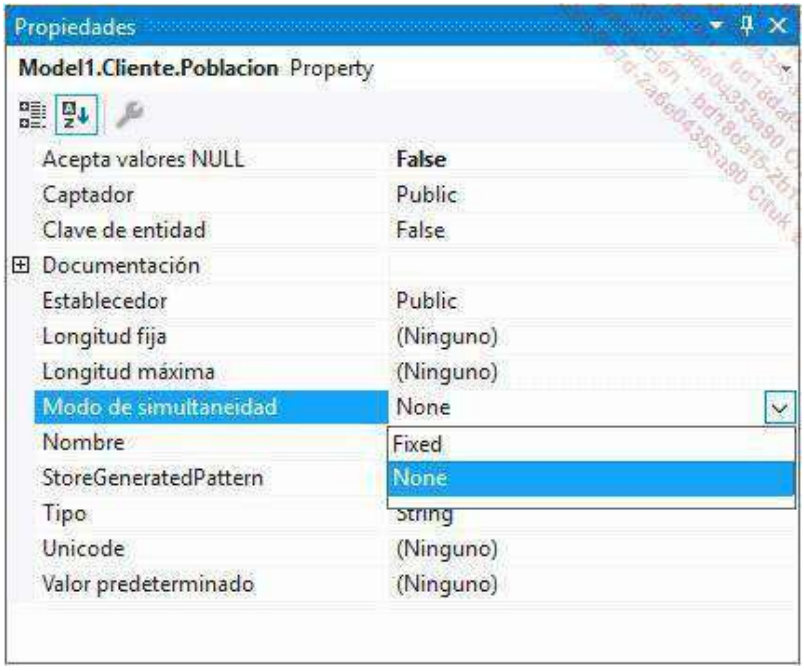
```
using (NorthwindDataContext context = new NorthwindDataContext())
{
    var cliente = (from c in context.Customers
                           where c.CustomerID == "JMART"
                           select c).First();

    context.Customers.Remove(cliente);

    context.SaveChanges();
}
```

c. Gestión de conflictos

Como hemos visto en el capítulo correspondiente a ADO.NET, uno de los principales problemas en el uso de bases de datos es la gestión de accesos concurrentes. Entity Framework, que no es más que una capa intermedia para ejecutar consultas SQL, presenta también este tipo de situaciones de conflicto durante su uso. Entity Framework incluye un mecanismo que permite gestionar estos problemas. Está compuesto por cuatro partes complementarias.

Configuración de la detección de conflictos

Tras la creación de las clases de mapeo con el diseñador visual es posible indicar para cada propiedad si se desea incluir en el mecanismo de gestión de conflictos o no y, en caso afirmativo, cómo debe gestionarse su presencia en este mecanismo. La ventana de propiedades permite editar el valor del campo **Modo de simultaneidad** (*Update Check*, en inglés) de cara a seleccionar el comportamiento adecuado.



Los valores disponibles para este campo son:

- **Fixed**: la detección de conflictos tiene en cuenta siempre a la propiedad.
- **None**: este campo no se utiliza jamás en la detección de conflictos. Es el **valor por defecto** para todas las propiedades.

Detección de conflictos

La detección de conflictos se realiza cuando los datos se envían a la base de datos. Esta etapa se produce cuando se invoca al método `SaveChanges` de `DbContext`. Si alguna de las peticiones de actualización no se ha podido ejecutar porque se ha detectado un registro diferente en la base de datos, se produce una excepción de tipo `DbUpdateException` a nivel de la instrucción del registro. Por tanto, es conveniente realizar cualquier llamada a `SaveChanges` en un bloque `try ... catch` para gestionar estos conflictos.

```
try
{
    dbContext.SaveChanges();
}
catch (DbUpdateConcurrencyException ex)
{
    // Tratamiento del error
}
```

Lectura de la información vinculada a un conflicto y resolución

Cuando se detectan uno o varios conflictos, la excepción aporta información que permite resolver los problemas encontrados. La colección `Entries` tiene el conjunto de registros en los que se ha producido un comportamiento inesperado.

Cada objeto de esta colección es de tipo `DbEntityEntry`. La propiedad `State` permite comprobar el estado de la entidad en cuestión (`Added`, `Deleted`, `Detached`, `Modified`, `Unchanged`). Solo con esta información es posible saber si no se ha podido agregar, eliminar o modificar una entidad. La propiedad `CurrentValues` de este tipo devuelve los valores del objeto para el que se ha detectado algún conflicto, mientras que la propiedad `OriginalValues` devuelve los valores del registro anteriores a la modificación local. El método `GetDatabaseValues` permite recuperar los datos anteriores a la instrucción que ha fallado. El uso de estos datos permite volver al estado inicial o construir un nuevo objeto que podrá sobreescribir los valores a nivel del origen de los datos.

Presentación

XML (*eXtensible Markup Language*) es un lenguaje de **descripción de datos** diseñado y estandarizado por el **W3C** (*World Wide Web Consortium*) cuya primera versión se corresponde con la especificación publicada a principios del año 1998.

El lenguaje XML es un lenguaje de etiquetado que permite describir la información de una manera estructurada, coherente y, sobre todo, portable. El formato está, en efecto, basado en texto bruto, lo que autoriza su lectura en cualquier tipo de soporte, sin verse alterado. Estas cualidades hacen de XML un formato preferente para el **intercambio de información** entre aplicaciones, incluso entre sistemas.

En un documento en formato XML, cada uno de los datos se representa mediante una **etiqueta** o un **atributo** situado dentro de un **árbol**. Las distintas etiquetas pueden estar anidadas de manera que exista una relación de parentesco entre dos elementos. Los atributos permiten vincular diversa información dentro de la misma etiqueta.

La estructura de un archivo XML lo sitúa en la misma categoría que el lenguaje HTML. Existen ciertas diferencias fundamentales respecto a este lenguaje:

- XML es un lenguaje extensible, a diferencia de HTML que no utiliza un conjunto fijo predefinido de etiquetas.
- HTML es un lenguaje concebido para la presentación de datos. Los documentos XML no tienen ninguna vocación concreta para realizar una representación gráfica. No hacen más que una descripción de los datos.

Estructura de un archivo XML

Antes de entrar a ver cómo manipular este formato con C# y el framework .NET, detallaremos los distintos elementos de la estructura de un archivo XML.

1. Componentes de un documento XML

Un documento XML está compuesto por los tipos de bloques siguientes:

Instrucciones de procesamiento

Las instrucciones de procesamiento permiten agregar información destinada al procesador XML o a cualquier otro programa en el cuerpo del documento. Estas instrucciones deben respetar una sintaxis particular:

```
<?programa instrucción ?>
```

La primera parte de esta instrucción de procesamiento indica el nombre del programa destinado a tratarla. La segunda parte proporciona la instrucción propiamente dicha.

Un documento XML contiene, generalmente, una instrucción de procesamiento especial que indica la versión del formato XML que utiliza así como el juego de caracteres utilizado para realizar la codificación del documento.

```
<?xml version="1.0" encoding="utf-8" ?>
```

Comentarios

Como con C#, es posible incluir comentarios en el código XML. Estos están destinados a los usuarios, generalmente para ayudarles a comprender la estructura o el contenido del documento. Se ignoran en cualquier programa que manipule el código.

La sintaxis de definición de comentarios XML es idéntica a la sintaxis utilizada para los comentarios HTML:

```
<!-- Este texto es un comentario, que ignora el procesador XML -->
```

Elementos

Los elementos están compuestos por una etiqueta de apertura y una etiqueta de cierre. Pueden contener datos en forma de cadena de caracteres, uno o varios elementos, o bien atributos. La sintaxis que permite crear un elemento XML es la siguiente:

```
<nombreElemento>contenido</nombreElemento>
```

Existen ciertas reglas que deben respetarse relativas a los elementos:

- Los nombres de los elementos no deben contener espacios, no pueden empezar por `xml`, por una cifra o por un signo de puntuación.
- Las etiquetas de inicio y de fin deben tener las mismas mayúsculas y minúsculas.

- Los nombres de los elementos deben comenzar tras el símbolo <, sin dejar espacio.
- Un documento XML debe contener como mínimo un elemento: el elemento raíz del documento.
- Si un elemento no tiene contenido, puede tener la forma `<nombreElemento />`, salvo si es el elemento raíz del documento.
- Ningún elemento puede estar situado al mismo nivel del árbol que el elemento raíz. Los elementos deben, por tanto, estar anidados en este elemento raíz.

```
<?xml version="1.0" encoding="utf-8"?>
<Clientes>
  <Cliente>
    <IdCliente>ALFKI</IdCliente>
    <Empresa>Alfreds Futterkiste</Empresa>
    <NombreContacto>Alfreds Maria Anders</NombreContacto>
    <Direccion>Obere Str. 57</Direccion>
    <CodigoPostal>12209</CodigoPostal>
    <Ciudad>Berlín</Ciudad>
    <UltimoPedido>
      <Productos>
        <Producto>
          <IdProducto>58</IdProducto>
          <Nombre>Naranjas de mesa</Nombre>
          <PrecioUnitario>1,25</PrecioUnitario>
          <Cantidad>40</Cantidad>
        </Producto>
        <Producto>
          <IdProducto>71</IdProducto>
          <Nombre>Aceite de oliva virgen extra</Nombre>
          <PrecioUnitario>2,50</PrecioUnitario>
          <Cantidad>20</Cantidad>
        </Producto>
      </Productos>
    </UltimoPedido>
  </Cliente>
</Clientes>
```

Atributos de los elementos

Los atributos representan datos almacenados a nivel de un elemento. Se sitúan en la etiqueta de apertura del elemento. Los atributos se definen mediante la siguiente sintaxis:

```
<nombreElemento atributo1="valor" atributo2="valor"></nombreElemento>
```

```
<?xml version="1.0" encoding="utf-8"?>
<Clientes>
  <Cliente IdCliente="ALFKI">
    <Empresa>Alfreds Futterkiste</Empresa>
    <NombreContacto>Alfreds Maria Anders</NombreContacto>
    <Direccion>Obere Str. 57</Direccion>
    <CodigoPostal>12209</CodigoPostal>
    <Ciudad>Berlín</Ciudad>
    <UltimoPedido Fecha="09/04/1998">
      <Productos>
        <Producto IdProducto="58">
```

```
          <Nombre>Naranjas de mesa</Nombre>
          <PrecioUnitario>1,25</PrecioUnitario>
          <Cantidad>40</Cantidad>
        </Producto>
        <Producto IdProducto="71">
          <Nombre>Aceite de oliva virgen extra</Nombre>
          <PrecioUnitario>2,50</PrecioUnitario>
          <Cantidad>20</Cantidad>
        </Producto>
      </Productos>
    </UltimoPedido>
  </Cliente>
</Clientes>
```

Espacios de nombres

Los espacios de nombres son identificadores que permiten distinguir varios elementos cuyo nombre es idéntico. Permiten evitar confusiones cuando varios elementos tienen el mismo nombre, pero un significado diferente. Este tipo de casos se da cuando se fusionan dos archivos XML.

Un espacio de nombres se define concatenando el atributo `xmlns` seguido del símbolo: más un prefijo para el espacio de nombres. El valor del atributo es un identificador único a menudo escrito como una URL. Este valor tiene que ser único para evitar cualquier conflicto en caso de que se transmita el documento a una aplicación en la que ya esté definido el elemento.

```
<nombreElemento xmlns:prefijo="identificador" />
```

El documento XML siguiente fusiona los objetos de tipo `Cliente` gestionados respectivamente por una aplicación web y una aplicación WPF en la tienda. Se distinguen en el documento resultante gracias a su espacio de nombres.

```
<?xml version="1.0" encoding="utf-8"?>
<Clientes xmlns:web="http://www.miempresa.com/clientes/web"
xmlns:tienda="http://www.miempresa.com/clientes/tienda">
  <web:Cliente Id="ALFKI">
    <Empresa>Alfreds Futterkiste</Empresa>
    <NombreContacto>Alfreds Maria Anders</NombreContacto>
    <Direccion>Obere Str. 57</Direccion>
    <CodigoPostal>12209</CodigoPostal>
    <Ciudad>Berlín</Ciudad>
    <UltimoPedido Fecha="09/04/1998">
      <Productos>
        <Producto IdProducto="58">
          <Nombre>Naranjas de mesa</Nombre>
          <PrecioUnitario>1,25</PrecioUnitario>
          <Cantidad>40</Cantidad>
        </Producto>
        <Producto IdProducto="71">
          <Nombre>Aceite de oliva virgen extra</Nombre>
          <PrecioUnitario>2,50</PrecioUnitario>
          <Cantidad>20</Cantidad>
        </Producto>
      </Productos>
    </UltimoPedido>
  </web:Cliente>
```

```
  <tienda:Cliente Id="684316" EsAnonimo="True">
    <Compra Fecha="12/04/1998" MedioDePago="Tarjeta">
      <Productos>
        <Producto IdProducto="11">
          <Nombre>Queso Cabrales</Nombre>
          <PrecioUnitario>21,00</PrecioUnitario>
          <Cantidad>2</Cantidad>
        </Producto>
        <Producto IdProducto="19">
          <Nombre>Galletas de chocolate</Nombre>
          <PrecioUnitario>9,20</PrecioUnitario>
          <Cantidad>7</Cantidad>
        </Producto>
      </Productos>
    </Compra>
  </tienda:Cliente>
</Clientes>
```

2. Documento bien formado y documento válido

El lenguaje XML define varios tipos de componentes que permiten describir una estructura de datos coherente. Pero esta noción no basta para definir la calidad de un documento XML. Existen dos nociones complementarias que se utilizan para ello: el documento bien formado y el documento válido.

Un documento bien formado es un documento XML que respeta las distintas reglas sintácticas del lenguaje. Estas reglas incluyen los distintos puntos que se han tratado hasta el momento.

Es necesario que el documento esté bien formado para que el procesador XML sea capaz de procesarlo.

La validez de un documento es un aspecto que puede verificarse mediante un DTD (*Document Type Definition*) o un esquema XSD (*Xml Schema Definition*) vinculados al archivo. Estos tipos de documentos proporcionan la descripción de la estructura de datos utilizada en el archivo XML. Si esta estructura no cumpliera las reglas fijadas por el DTD o el esquema XSD, el archivo no sería válido.

Manipular un documento XML

El framework .NET expone dos API cuyo objetivo es manipular documentos XML.

El **modelo DOM** existe desde los inicios del framework .NET y está basado en las recomendaciones de W3C. Cada etiqueta, atributo o contenido textual que forman parte del documento XML se considera como un nodo de una estructura jerárquica. La búsqueda de un nodo mediante esta API puede realizarse manualmente o mediante la navegación **XPath**. Las funcionalidades del modelo DOM se implementan en los tipos del espacio de nombres `System.Xml`.

La aparición de la versión 3.5 del framework .NET introduce **LINQ to XML**, que utiliza por su parte un enfoque más moderno. Las etiquetas, atributos o espacios de nombres se consideran como elementos que pueden existir independientemente, lo que permite procesar fácilmente fragmentos de código XML. Es posible interrogar a estos elementos de la misma manera que se interroga a una base de datos mediante consultas LINQ. Los tipos asociados a esta API se implementan en el espacio de nombres `System.Xml.Linq`.

Los ejemplos expuestos en esta sección están realizados a partir del archivo `pedidos.xml` cuyo contenido es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<Clientes web="http://www.miempresa.com/clientes/web">
  <web:Cliente Id="ALFKI">
    <Empresa>Alfreds Futterkiste</Empresa>
    <NombreContacto>Alfreds Maria Anders</NombreContacto>
    <Direccion>Obere Str. 57</Direccion>
    <CodigoPostal>12209</CodigoPostal>
    <Ciudad>Berlín</Ciudad>
    <UltimoPedido Fecha="09/04/1998">
      <Productos>
        <Producto>
          <Nombre>Naranjas de mesa</Nombre>
          <PrecioUnitario>1,25</PrecioUnitario>
          <Cantidad>40</Cantidad>
        </Producto>
        <Producto>
          <Nombre>Aceite de oliva virgen extra</Nombre>
          <PrecioUnitario>2,50</PrecioUnitario>
          <Cantidad>20</Cantidad>
        </Producto>
      </Productos>
    </UltimoPedido>
  </web:Cliente>
</Clientes>
```

Este archivo está disponible para su descarga desde la página Información.

1. Uso de DOM

La primera etapa en el uso de DOM consiste en cargar un documento XML en memoria. Se representa mediante un objeto de tipo `XmlDocument` que contiene el conjunto del árbol del documento.

La lectura y la carga de un archivo XML se realizan mediante el uso de los métodos `Load` o `LoadXml` del tipo `XmlDocument`.

```
XmlDocument doc = new XmlDocument();
doc.Load("pedidos.xml");
```

```
//La cadena de caracteres debe contener una estructura de documento XML
string xml = "...";
doc.LoadXml(xml);
```

No es obligatorio crear este objeto en memoria a partir de un archivo. La API DOM define varias clases y métodos que permiten generar un documento XML mediante código C#. El siguiente código crea un objeto `XmlDocument` que contiene el mismo árbol que el archivo `pedidos.xml`.

```
XmlDocument doc = new XmlDocument();

//Creación del nodo de declaración XML
XmlDeclaration declaracionXml = doc.CreateXmlDeclaration("1.0",
"utf-8", null);
doc.AppendChild(declaracionXml);

//Creación del nodo Clientes
XmlElement raiz = doc.CreateElement("Clientes");
//Creación del atributo que define el namespace web
raiz.SetAttribute("xmlns:web",
"http://www.miempresa.com/clientes/web");
doc.AppendChild(raiz);

//Creación del nodo Cliente e indicación del espacio de nombres asociado
XmlElement cliente1 = doc.CreateElement("Cliente",
"http://www.miempresa.com/clientes/web");
cliente1.Prefix = "web";
raiz.AppendChild(cliente1);
//Creación del atributo Id del nodo Cliente
cliente1.SetAttribute("Id", "ALFKI");

//Creación del nodo Empresa
XmlElement empresa = doc.CreateElement("Empresa");
//Creación del nodo texto que contiene el valor de Empresa
XmlText valorEmpresa = doc.CreateTextNode("Alfreds
Futterkiste");
empresa.AppendChild(valorEmpresa);
cliente1.AppendChild(empresa);

//Creación del nodo NombreContacto
XmlElement nombreContacto = doc.CreateElement("NombreContacto");
//Creación del nodo texto que contiene el valor de NombreContacto
XmlText valorNombreContacto = doc.CreateTextNode("Alfreds Maria
Anders");
nombreContacto.AppendChild(valorNombreContacto);
```

```
cliente1.AppendChild(NombreContacto);
XmlElement direccion =  doc.CreateElement("Direccion");
XmlText valorDireccion = doc.CreateTextNode("Obere Str. 57");
direccion.AppendChild(valorDireccion);
cliente1.AppendChild(direccion);
XmlElement codigoPostal = doc.CreateElement("CodigoPostal");
XmlText valorCodigoPostal = doc.CreateTextNode("12209");
codigoPostal.AppendChild(valorCodigoPostal);
cliente1.AppendChild(CodigoPostal);
XmlElement ciudad = doc.CreateElement("Ciudad");
XmlText valorCiudad = doc.CreateTextNode("Berlín");
ciudad.AppendChild(valorCiudad);
cliente1.AppendChild(ciudad);

//Creación del nodo UltimoPedido
XmlElement ultimoPedido =
doc.CreateElement("UltimoPedido");
cliente1.AppendChild(UltimoPedido);
//Creación del atributo Fecha del nodo UltimoPedido
ultimoPedido.SetAttribute("Fecha", "09/04/1998");

//Creación del nodo Productos
XmlElement productos = doc.CreateElement("Productos");
ultimoPedido.AppendChild(Productos);
//Creación del primer nodo Producto
XmlElement producto1 = doc.CreateElement("Producto");
productos.AppendChild(producto1);
//Creación del nodo Nombre
XmlElement nombre1 = doc.CreateElement("Nombre");
XmlText valorNombre1 = doc.CreateTextNode("Naranjas de mesa");
nombre1.AppendChild(valorNombre1);
producto1.AppendChild(nombre1);
//Creación del nodo PrecioUnitario
XmlElement precioUnitario1 = doc.CreateElement("PrecioUnitario");
XmlText valorPrecioUnitario1 = doc.CreateTextNode("1,25");
precioUnitario1.AppendChild(valorPrecioUnitario1);
producto1.AppendChild(PrecioUnitario1);
//Creación del nodo Cantidad
XmlElement cantidad1 = doc.CreateElement("Cantidad");
XmlText valorCantidad1 = doc.CreateTextNode("40");
cantidad1.AppendChild(valorCantidad1);
producto1.AppendChild(cantidad1);

//Creación del segundo nodo Producto
XmlElement producto2 = doc.CreateElement("Producto");
productos.AppendChild(producto2);
//Creación del nodo Nombre
XmlElement nombre2 = doc.CreateElement("Nombre");
XmlText valorNombre2 = doc.CreateTextNode("Aceite de oliva virgen extra");
nombre2.AppendChild(valorNombre2);
producto2.AppendChild(nombre2);
//Creación del nodo PrecioUnitario
XmlElement precioUnitario2 = doc.CreateElement("PrecioUnitario");
XmlText valorPrecioUnitario2 = doc.CreateTextNode("2,50");
precioUnitario2.AppendChild(valorPrecioUnitario2);
```

```
producto2.AppendChild(PrecioUnitario2);
//Creación del nodo Cantidad
XmlElement cantidad2 = doc.CreateElement("Cantidad");
XmlText valorCantidad2 = doc.CreateTextNode("20");
cantidad2.AppendChild(valorCantidad2);
producto2.AppendChild(cantidad2);
```

Analicemos los distintos elementos utilizados en el ejemplo.

`XmlElement` representa una etiqueta XML para la que es necesario proporcionar un nombre. La manera más sencilla para crear un objeto de este tipo es invocar al método `CreateElement` del objeto `XmlDocument` que contenga el árbol XML. La propiedad `Prefix` (utilizada aquí sobre `cliente1`) del tipo `XmlElement` permite definir el prefijo del espacio de nombres que se desea utilizar para el elemento.

La creación de atributos para un objeto `XmlElement` se realiza agregando objetos de tipo `XmlAttribute` a su colección `Attributes`. Es, generalmente, más apropiado utilizar su método `SetAttribute`, más sencillo y más legible. Este método recibe dos parámetros que representan, respectivamente, el nombre del atributo así como su valor. Utilizando este último método sobre el elemento raíz se agrega al documento el espacio de nombres `web`.

La anidación de los distintos componentes del documento se consigue mediante el método `AppendChild`. Se utiliza sobre el nodo destinado a ser el padre del nodo que se pasa como parámetro. Este método se define sobre la clase `XmlNode`, de modo que todos sus tipos derivados puedan utilizarla. Encontramos, entre ellos, los tipos `XmlDocument`, `XmlElement` y `XmlAttribute`.

El tipo `XmlText` es ligeramente distinto puesto que no puede contener otros elementos. Es el verdadero contenedor de los datos, mientras que los demás solo permiten definir la estructura del documento. Los objetos de este tipo se crean mediante una llamada al método `CreateTextNode` del objeto `XmlDocument`. El parámetro que se pasa a esta función es el dato que debe escribirse.

Una vez cargado o generado el documento, es posible buscar ciertos elementos a partir de su nombre o su identificador.

El método `GetElementsByTagName` de la clase `XmlNode` se utiliza para encontrar todos los nodos XML cuyo nombre se corresponda con el valor que se pasa como parámetro. Realiza una búsqueda en toda la arborescencia hija de un elemento. La lista de nodos devuelta puede contener elementos de cualquier nivel: nodos hijos, nietos o incluso de un nivel inferior.

El siguiente ejemplo de código muestra por pantalla el precio unitario de cada uno de los productos pedidos.

```
XmlNodeList precioUnitarios =
doc.GetElementsByTagName("PrecioUnitario");
foreach (XmlNode nodo in  precioUnitarios)
{
   Console.WriteLine(nodo.InnerText);
}
```

Cuando el documento contiene un DTD, es posible enumerar los elementos que se corresponden con un identificador mediante el método `GetElementById`. El DTD debe definir, para ello, uno o varios atributos de tipo `ID`.

La inserción del siguiente DTD entre la declaración XML y el nodo raíz del documento define el atributo `Id` de los elementos de tipo `web:Cliente` como identificador para la etiqueta.

```
<!DOCTYPE Clientes [
  <!ELEMENT Clientes ANY>
  <!ELEMENT web:Cliente ANY>
  <!ATTLIST web:Cliente Id ID #REQUIRED>
]>
```

Una vez agregada esta definición, es posible buscar y mostrar el nodo cuyo identificador es ALFKI.

```
XmlNode clienteALFKI = doc.GetElementById("ALFKI");
Console.WriteLine(clienteALFKI.OuterXml);
```

Una vez seleccionado un nodo, es bastante sencillo modificar alguno de sus atributos o agregarle uno o varios nodos hijos.

```
//Modificación de la fecha y agregación de un nodo Comercial
//para cada uno de los nodos UltimoPedido
XmlNodeList pedidos =
doc.GetElementsByTagName("UltimoPedido");
foreach (XmlElement nodo in pedidos)
{
    nodo.Attributes["Fecha"].Value = "24/10/2001";

    XmlElement comercial = doc.CreateElement("Comercial");
    XmlText valorComercial = doc.CreateTextNode("Steven Buchanan");
    comercial.AppendChild(valorComercial);
    nodo.AppendChild(comercial);
}
```

El registro del documento con las modificaciones aportadas debe realizarse invocando al método `Save` del objeto `XmlDocument`. Sin esta operación, cualquier modificación se pierde, pues el documento se trata únicamente en memoria.

```
doc.Save("pedidosModificados.xml");
```

2. Uso de XPath

XPath tiene como objetivo localizar uno o varios elementos de la misma manera que el explorador de Windows permite direccionar un archivo particular. La implementación de la tecnología XPath en el framework .NET se incluye en el espacio de nombres `System.Xml.XPath`.

La primera etapa en el uso de XPath se corresponde con la instanciación de un objeto de tipo `XPathNavigator`. Este objeto proporciona la capacidad de direccionar los elementos de un documento XML.

```
XmlDocument doc = new XmlDocument();
doc.Load("pedidos.xml");
```

```
XPathNavigator navegador = doc.CreateNavigator();
```

Uno de los métodos de este objeto, `Select`, ofrece la posibilidad de enumerar los elementos que se corresponden con la ruta XPath que se le pasa como parámetro.

```
XPathNavigator navegador = doc.CreateNavigator();

XPathNodeIterator productos =
navegador.Select("/Clientes/Cliente/UltimoPedido/Productos/
Producto");

while (productos.MoveNext())
{
   Console.WriteLine(productos.Current. OuterXml);
}
```

En nuestro caso, este código no muestra nada por pantalla, lo cual es normal: los nodos `Cliente` y `web:Cliente` son diferentes. Es necesario, por tanto, agregar algunos elementos para que el objeto `XPathNavigator` pueda devolver los elementos correspondientes.

La ruta que se pasa como parámetro al método `Select` debe contener el nombre plenamente cualificado de cada uno de los tipos de nodos. En nuestro ejemplo, esto supone utilizar `web:Cliente` en lugar de `Cliente`.

Para que el navegador XPath pueda comprender y encontrar este nombre cualificado es necesario proporcionarle un objeto de tipo `XmlNamespaceManager` al que se le agrega la definición del espacio de nombres `web`.

El código funcional es el siguiente:

```
XPathNavigator navegador = doc.CreateNavigator();
XmlNamespaceManager manager = new
XmlNamespaceManager(navegador.NameTable);
manager.AddNamespace("web",
"http://www.miempresa.com/clientes/web");

XPathNodeIterator productos =
navegador.Select("/Clientes/web:Cliente/UltimoPedido/
Productos/Producto", manager);

while (productos.MoveNext())
{
    Console.WriteLine(productos.Current.OuterXml);
}
```

La modificación de los valores no es mucho más complicada, pues basta con aplicar el método `SetValue` a un elemento devuelto por la función `Select`. Es posible realizar un aumento del 10% sobre el precio de cada producto pedido de la siguiente manera:

```
XPathNodeIterator precioUnitarios =
navegador.Select("/Clientes/web:Cliente/UltimoPedido/Productos/
```

```
Producto/PrecioUnitario", manager);

while (precioUnitarios.MoveNext())
{
    double valor = double.Parse(precioUnitarios.Current.Value);
    valor = valor * 1.10;
    precioUnitarios.Current.SetValue(valor.ToString());
}
```

Tras la ejecución de esta sección de código, cada uno de los nodos `PrecioUnitario` del documento XML ha cambiado su valor.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Clientes[
  <!ELEMENT Clientes ANY>
  <!ELEMENT web:Cliente ANY>
  <!ATTLIST web:Cliente Id ID #REQUIRED>
]>
<Clientes xmlns:web="http://www.miempresa.com/clientes/web">
  <web:Cliente Id="ALFKI">
    <Empresa>Alfreds Futterkiste</Empresa>
    <NombreContacto>Alfreds Maria Anders</NombreContacto>
    <Direccion>Obere Str. 57</Direccion>
    <CodigoPostal>12209</CodigoPostal>
    <Ciudad>Berlín</Ciudad>
    <UltimoPedido Fecha="09/04/1998">
      <Productos>
        <Producto>
          <Nombre>Naranjas de mesa</Nombre>
          <PrecioUnitario>1,375</PrecioUnitario>
          <Cantidad>40</Cantidad>
        </Producto>
        <Producto>
          <Nombre>Aceite de oliva virgen extra</Nombre>
          <PrecioUnitario>2,75</PrecioUnitario>
          <Cantidad>20</Cantidad>
        </Producto>
      </Productos>
    </UltimoPedido>
  </web:Cliente>
</Clientes>
```

3. Uso de LINQ to XML

La aparición de LINQ en el framework .NET aporta numerosas mejoras en la manipulación de datos propios de colecciones locales o de bases de datos. Con el proveedor LINQ para SQL Server, Microsoft proporciona también un proveedor para los documentos XML, permitiendo así manipular de manera más sencilla los datos registrados usando este formato.

El proveedor LINQ to XML contiene nuevos tipos que simplifican la creación de documentos XML e integran, también, las funcionalidades necesarias para realizar consultas sobre los datos que contienen. Se encuentran en el espacio

de nombres `System.Xml.Linq`.

La primera etapa en el uso de LINQ to XML consiste en cargar un documento XML en memoria. Esta operación se realiza utilizando el método estático `Load` de la clase `XDocument`.

```
XDocument doc = XDocument.Load("pedidos.xml");
```

Como con la API DOM, es posible crear un árbol XML directamente en memoria. El código C# siguiente instancia y asigna valor a un objeto `XDocument`.

```
XNamespace webNamespace = "http://www.miempresa.com/clientes/web";
XDocument doc =
   new XDocument(
      new XDeclaration("1.0", "utf-8", null),
      new XElement("Clientes",
         new XAttribute(XNamespace.Xmlns + "web", webNamespace),
         new XElement(webNamespace + "Cliente",
            new XAttribute("Id", "ALFKI"),
            new XElement("Empresa", "Alfreds Futterkiste"),
            new XElement("NombreContacto", "Alfreds Maria Anders"),
               new XElement("Direccion", "Obere Str. 57"),
               new XElement("CodigoPostal", "12209"),
               new XElement("Ciudad", "Berlín"),
               new XElement("UltimoPedido",
                   new XAttribute("Fecha", "09/04/1998"),
                   new XElement("Productos",
                      new XElement("Producto",
                          new XElement("Nombre", "Naranjas de mesa"),
                          new XElement("PrecioUnitario", "13,25"),
                          new XElement("Cantidad", "40")),
                      new XElement("Producto",
                         new XElement("Nombre", "Aceite de oliva virgen extra"),
                          new XElement("PrecioUnitario", "21,50"),
                          new XElement("Cantidad", "20")))))));
```

El código LINQ es **cuatro veces más corto** que el código DOM y genera la misma estructura de árbol, manteniendo una **expresividad** que la API DOM es incapaz de igualar. Otro aspecto positivo está relacionado con el **número de variables necesarias** para crear este documento: dos, con LINQ, ¡frente a una treintena en el caso anterior!

Un documento XML se representa con LINQ mediante un objeto de tipo `XDocument`. Define un constructor que recibe un número variable de parámetros, lo que permite agregar un árbol hijo entero desde su creación. El principio es el mismo para el tipo `XElement`, que representa un nodo del árbol. Este modo de funcionamiento permite encadenar la creación de objetos, lo cual genera un código más conciso y más expresivo.

La lectura de información en este árbol se ve, también, simplificada. La escritura de consultas LINQ para la búsqueda aprovecha ciertos métodos presentes en la clase `XElement`:

- `Descendants` devuelve la lista de nodos del árbol hijo del nodo en curso.
- `Elements` devuelve únicamente los hijos directos del nodo en curso.
- `AncestorsAndSelf` devuelve la lista de nodos encontrados recorriendo ascendentemente la jerarquía desde el

nodo en curso hasta el nodo raíz. El resultado incluye el nodo en curso.

- `Attributes` enumera los atributos del nodo en curso.
- `DescendantsAndSelf` devuelve la lista de nodos del árbol hijo del nodo en curso y los incluye en el resultado.

El siguiente código realiza la búsqueda de los nodos cuyo nombre es `PrecioUnitario`.

```
var precioUnitarios =
    from nodo in doc.Descendants("PrecioUnitario")
    select nodo;

foreach (XElement nodo in precioUnitarios)
{
    Console.WriteLine(nodo.Value);
}
```

Es posible, también, buscar elementos cuyo atributo tenga un valor particular. El siguiente ejemplo muestra cómo buscar el primer nodo cuyo atributo `Id` tenga el valor "`ALFKI`".

```
var clienteALFKI =
    (from nodoCliente in doc.Descendants(webNamespace + "Cliente")
     where nodoCliente.Attribute("Id").Value == "ALFKI"
     select nodoCliente).FirstOrDefault();

Console.WriteLine(clienteALFKI);
```

Es posible modificar el valor de algún atributo o elemento asignando un nuevo valor a la propiedad `Value` del elemento. El siguiente código modifica las fechas de los pedidos y los precios unitarios registrados en el documento.

```
var nodosPedidoOPrecio =
    from nodo in doc.Descendants()
    where nodo.Name == "UltimoPedido"
       || nodo.Name == "PrecioUnitario"
    select nodo;

foreach (XElement nodo in nodosPedidoOPrecio)
{
    if (nodo.Name == "UltimoPedido")
    {
        nodo.Attribute("Fecha").Value = "28/05/2014";
    }
    if (nodo.Name == "PrecioUnitario")
    {
        var nuevoValor = double.Parse(nodo.Value) * 1.10;
        nodo.Value = nuevoValor.ToString();
    }
}
```

Es posible guardar las modificaciones como se hace con la API DOM, invocando al método `Save` sobre el objeto

XDocument.

```
doc.Save("pedidos modificados con Linq.xml");
```

El contenido del archivo es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<Clientes xmlns:web="http://www.miempresa.com/clientes/web">
  <web:Cliente Id="ALFKI">
    <Empresa>Alfreds Futterkiste</Empresa>
    <NombreContacto>Alfreds Maria Anders</NombreContacto>
    <Direccion>Obere Str. 57</Direccion>
    <CodigoPostal>12209</CodigoPostal>
    <Ciudad>Berlín</Ciudad>
    <UltimoPedido Fecha="28/05/2014">
      <Productos>
        <Producto>
          <Nombre>Naranjas de mesa</Nombre>
          <PrecioUnitario>1,375</PrecioUnitario>
          <Cantidad>40</Cantidad>
        </Producto>
        <Producto>
          <Nombre>Aceite de oliva virgen extra</Nombre>
          <PrecioUnitario>2,75</PrecioUnitario>
          <Cantidad>20</Cantidad>
        </Producto>
      </Productos>
    </UltimoPedido>
  </web:Cliente>
</Clientes>
```

Introducción

El final del ciclo de desarrollo es una fase crítica para cualquier proyecto. El código fuente de la aplicación se ha completado, probado, depurado y, eventualmente, optimizado, pero todavía es necesario abordar una problemática: ¿cómo proveer esta aplicación a los usuarios?

Visual Studio permite utilizar dos tecnologías dedicadas a esta fase de despliegue:

- **Windows Installer**: la aplicación y sus dependencias se incorporan en un archivo ejecutable, llamado generalmente `setup.exe`, que el usuario ejecuta.
- **ClickOnce**: los archivos de la aplicación están centralizados y el usuario la ejecuta o instala desde la ubicación remota donde se encuentran almacenados los archivos.

Windows Installer

Windows Installer es un servicio de despliegue disponible para todos los sistemas operativos de Microsoft. El principio de funcionamiento de este servicio consiste en la agrupación de todos los componentes de una aplicación en un único archivo ejecutable que puede distribuirse fácilmente a los usuarios.

Cuando una aplicación se instala mediante Windows Installer, muchas operaciones que identifican las operaciones realizadas se almacenan en el sistema. Esta información es relativa, en particular, a los archivos copiados, las modificaciones realizadas en el registro de Windows o el registro de componentes externos necesarios para el correcto funcionamiento de la aplicación. Esta información resulta esencial: permiten al servicio de Windows Installer realizar convenientemente la desinstalación de la aplicación, si se solicita.

Cada una de las modificaciones realizadas durante la instalación se aborda durante la desinstalación de cara a no dejar ninguna traza. No obstante, Windows Installer verifica la presencia de componentes compartidos antes de realizar cualquier eliminación para evitar afectar al comportamiento de otras aplicaciones que podrían estar utilizando estos componentes.

Cuando una aplicación utiliza Windows Installer, su proceso de instalación es transaccional, es decir, o bien la instalación funciona bien en su conjunto, o bien se anula completamente. Esta anulación supone la eliminación completa de los archivos que se hubieran copiado así como la restauración del estado original del sistema. Este último queda, siempre, en un estado coherente.

Windows Installer proporciona también un mecanismo de reparación de una aplicación que remplaza los archivos corruptos o faltantes.

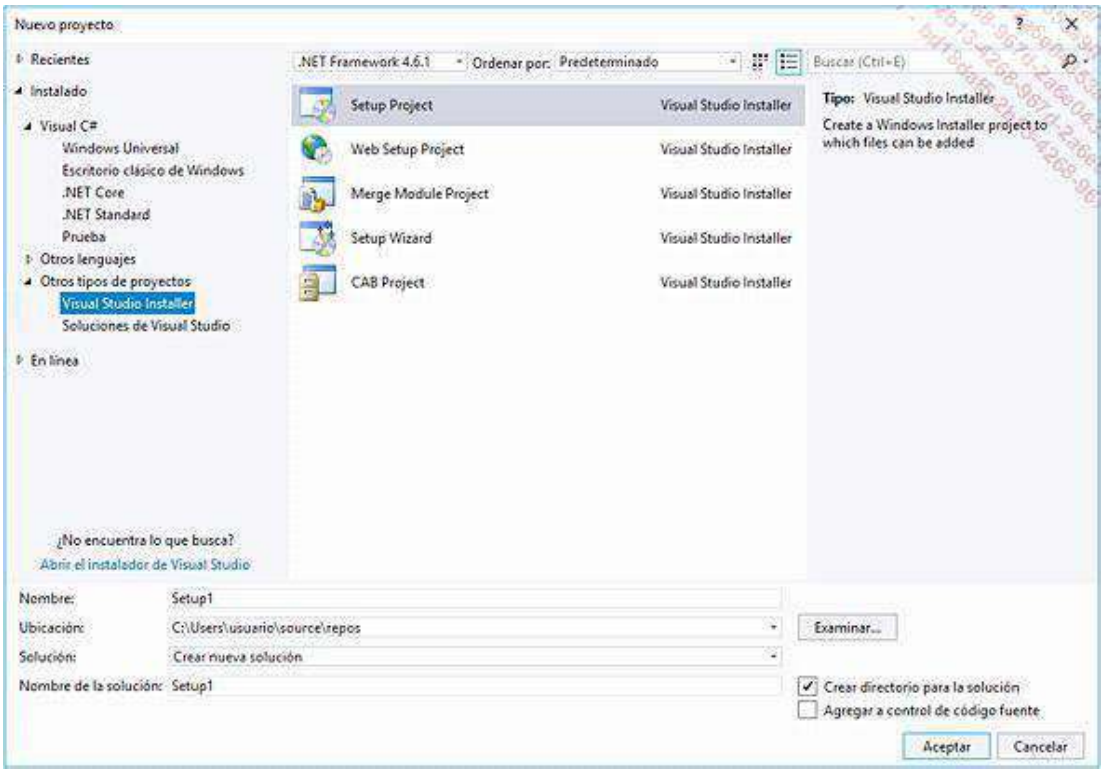
La creación de un ejecutable Windows Installer se realiza mediante un tipo de proyecto específico de Visual Studio. De todos modos, los modelos de proyecto asociados no están disponibles en la instalación de Visual Studio 2017. Microsoft pone a disposición del usuario una extensión dedicada al soporte de este tipo de proyectos en Visual Studio Marketplace: Microsoft Visual Studio Installer Projects.

Se puede descargar rápidamente (menos de 5 MB) de la dirección: https://marketplace.visualstudio.com/items?itemName=VisualStudioProductTeam.MicrosoftVisualStudio2017InstallerProjects

Antes de empezar a utilizar esta extensión es necesario reiniciar Visual Studio.

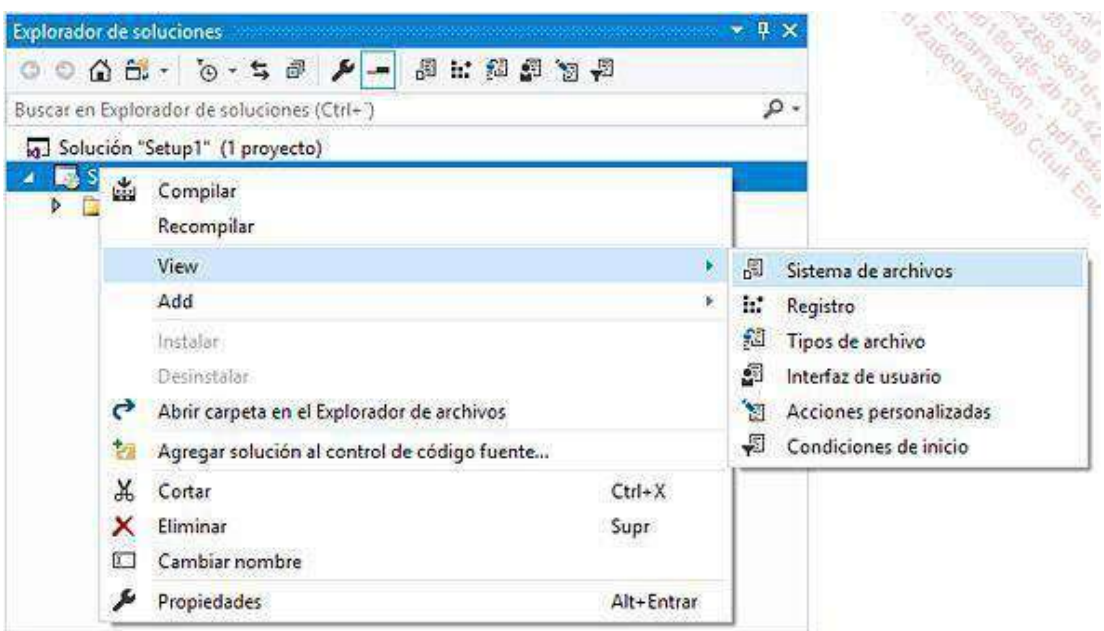
1. Creación de un proyecto de instalación

Los proyectos de instalación se crean de la misma manera que cualquier otro tipo de proyecto en Visual Studio. En el cuadro de diálogo **Nuevo proyecto**, seleccione **Otros tipos de proyectos - Visual Studio Installer** y, a continuación, **Setup Project**. Este modelo de proyecto está adaptado para el despliegue de aplicaciones de escritorio. El modelo **Setup Wizard** permite hacer lo mismo, pero con una fase preliminar de tipo "asistente de creación".



Es perfectamente posible generar un programa de instalación para una aplicación sin formar parte de la misma solución, pero se recomienda conservar cierta coherencia situando los proyectos que forman una aplicación y el proyecto de instalación correspondiente dentro de la misma solución. Además, esto permite asegurar que se pueden integrar todas las dependencias asociadas al programa de instalación.

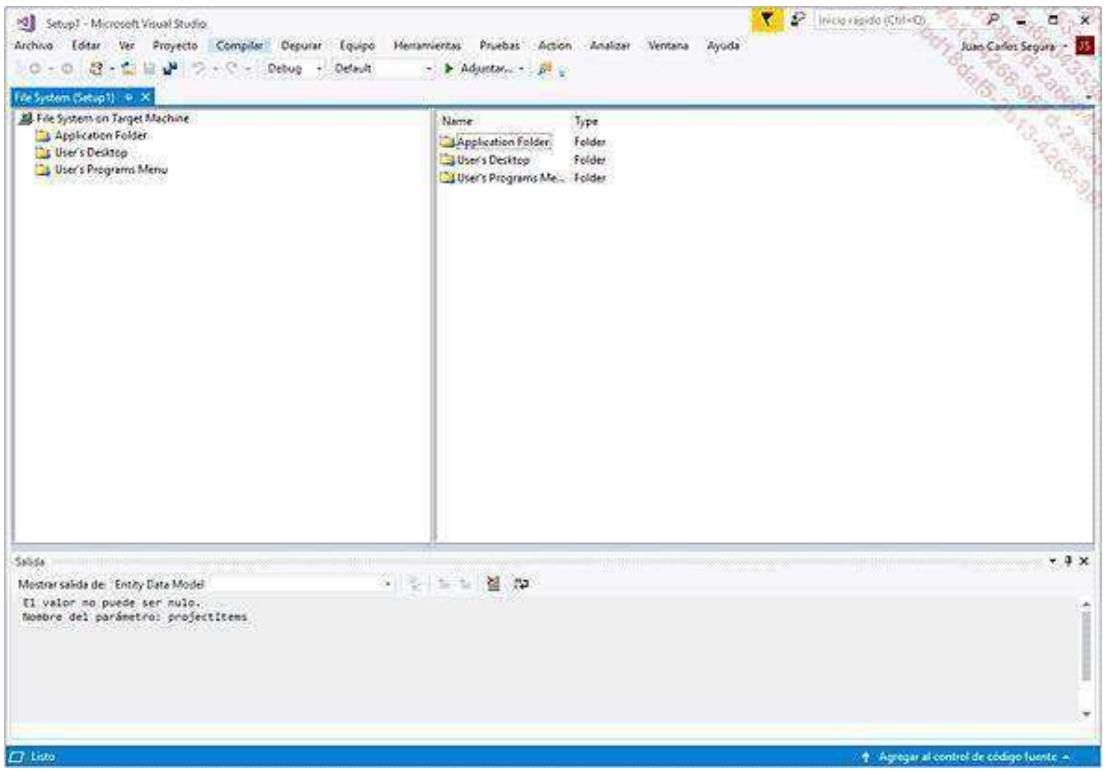
Durante la instalación se pueden realizar diferentes operaciones. Es posible acceder a ellas mediante el explorador de soluciones de Visual Studio, más concretamente a través del menú contextual del proyecto de instalación.



a. Operaciones sobre el sistema de archivos

Después de la creación del proyecto, en Visual Studio se abre una ventana de tipo "explorador de archivos" que permite visualizar y modificar los elementos que se crearán a nivel de sistema de archivos durante la fase de

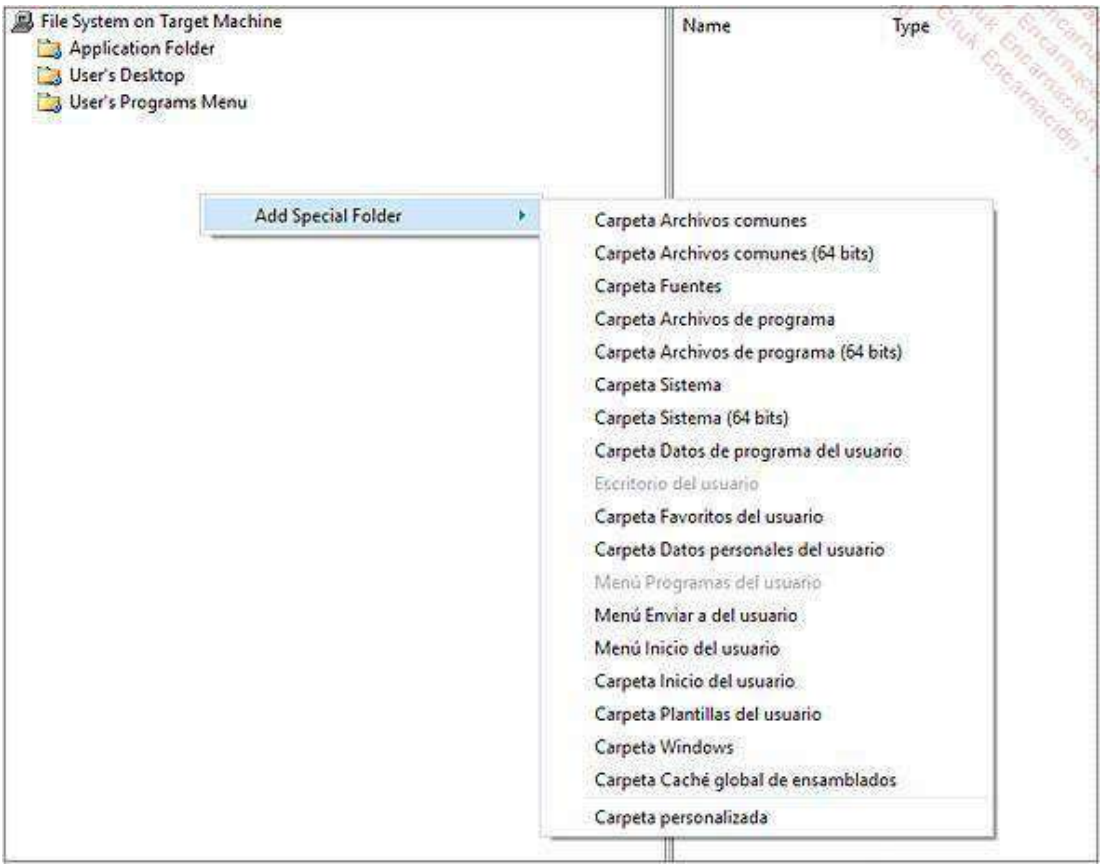
instalación.



Esta pantalla muestra por defecto carpetas del sistema de archivos del usuario susceptibles de ser asignadas por la instalación.

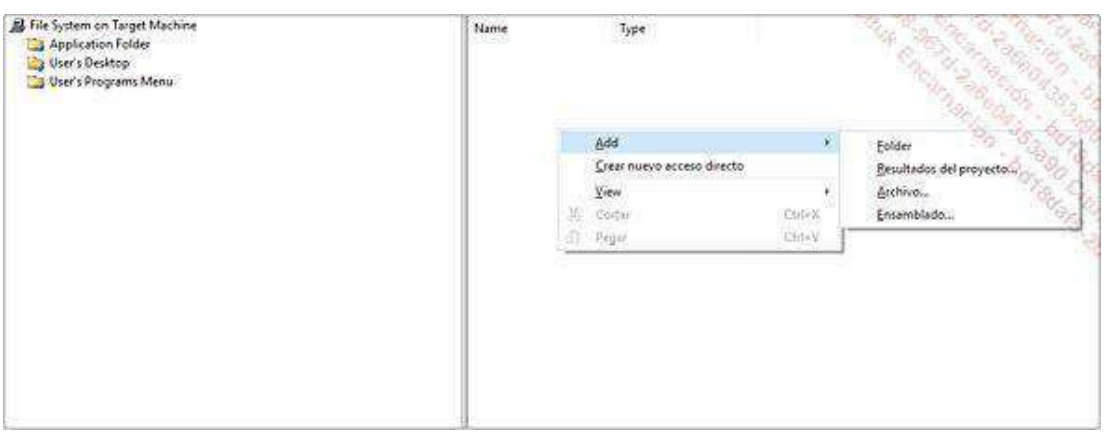
- La carpeta de instalación de la aplicación, que debe contener los archivos ejecutables, librerías y recursos de la aplicación.
- El escritorio del usuario, que podría contener uno o varios accesos directos.
- El menú **Inicio** del usuario, que podría contener diferentes entradas que corresponden al programa instalado así como elementos vinculados a él (archivos de ayuda, enlace a un sitio web de soporte, etc.).

Es posible añadir otras carpetas a esta lista. Haciendo clic con el botón derecho en la zona izquerda del explorador se abre un menú que permite hacerlo.

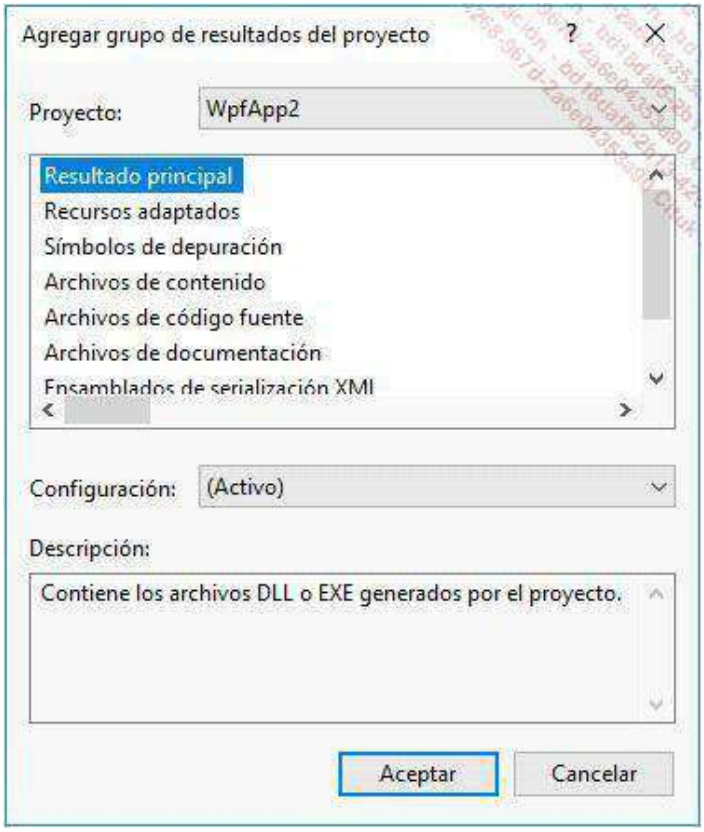


La instalación modifica el contenido de cada una de las carpetas del menú contextual cuando se hace clic con el botón derecho en la zona derecha del explorador. Se pueden agregar diferentes elementos:

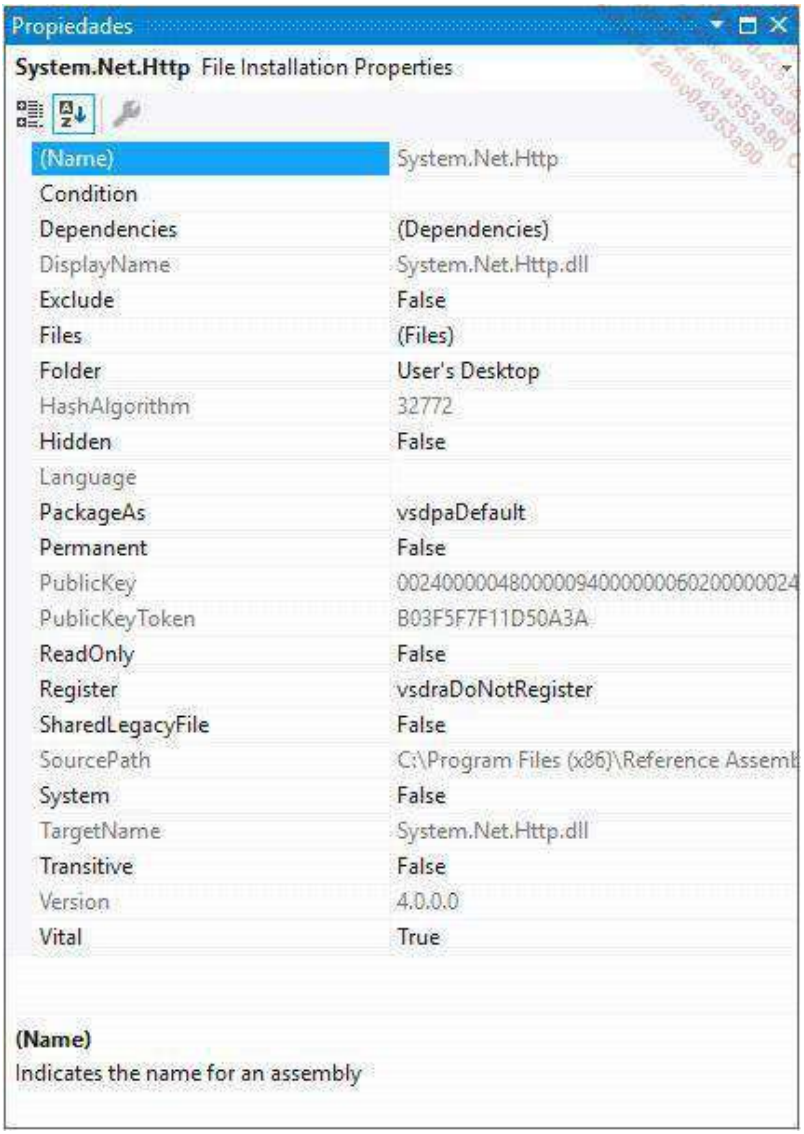
- **Folder** (carpeta): una carpeta para organizar los diferentes elementos que se pueden agregar a la carpeta raíz.
- **Resultados del proyecto**: los archivos creados por la generación de un proyecto (DLL, exe, archivos de configuración…).
- **Archivo**: un archivo independiente de la salida del proyecto, como una documentación o un archivo que describa la evolución de la aplicación.
- **Ensamblado**: un ensamblado .NET del que depende la aplicación.



La selección de una salida de proyecto permite definir qué archivos del proyecto se deben integrar. Para cada una de las opciones posibles, hay una descripción que explica qué elementos se incluyen. En esta lista es posible realizar una selección múltiple con las teclas [Ctrl] y [Mayús].



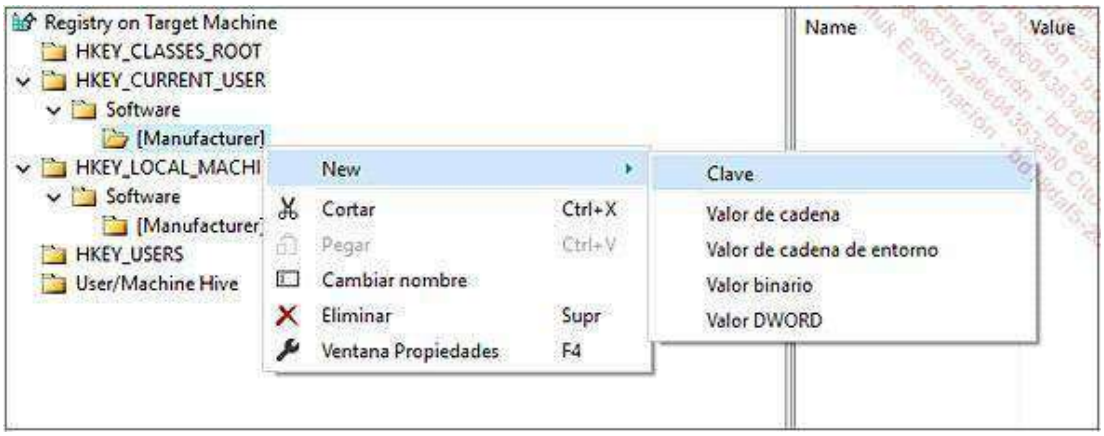
Cada elemento añadido al sistema tiene diferentes propiedades que se pueden editar con la ventana **Propiedades** de Visual Studio. También se puede especificar que un archivo se debe instalar como archivo de caché o archivo del sistema, o incluso indicar una condición para la instalación de este elemento. Este último punto se explica más adelante en este capítulo.



b. Operaciones en el registro de la máquina donde se hace la instalación

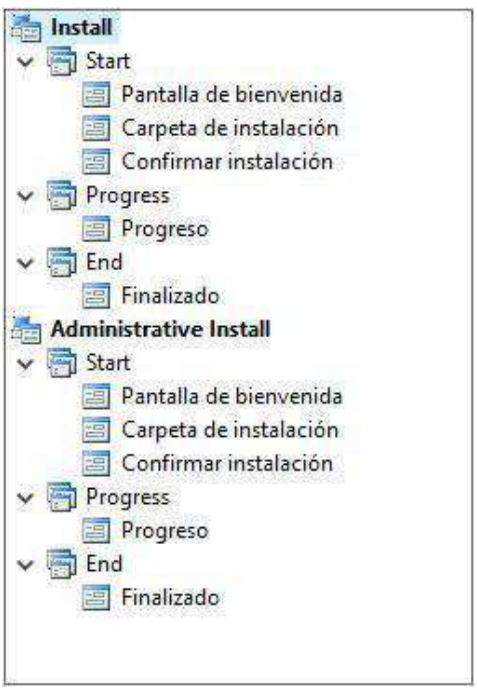
El registro de Windows es un elemento que se utiliza habitualmente para registrar la información de configuración. Es una base de datos que utiliza el sistema operativo y las aplicaciones instaladas. Está estructurada en diferentes secciones (hives) que están representadas por defecto en la interfaz de Visual Studio.

Haciendo clic con el botón derecho en uno de los elementos del árbol se abre un menú contextual que permite crear claves de registro o valores para una clave.

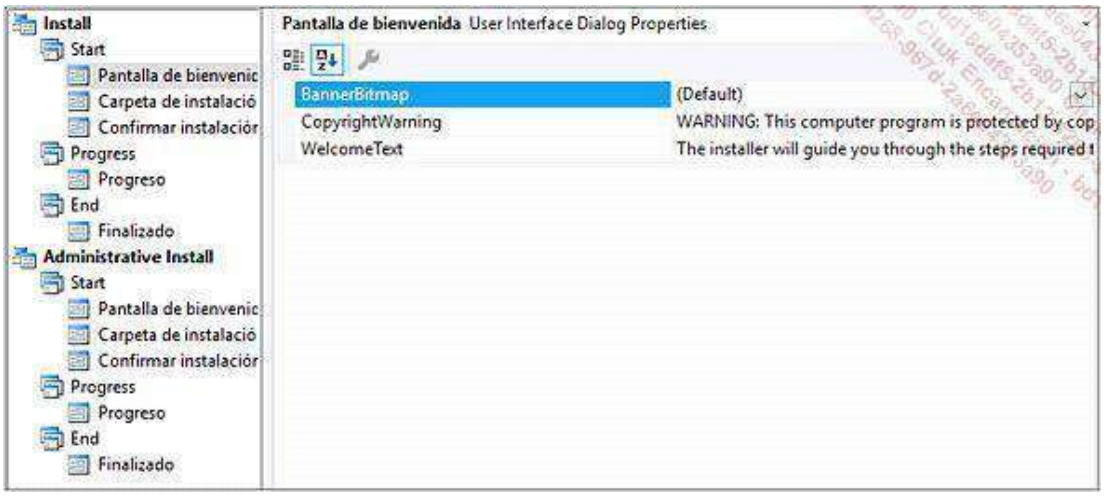


c. Configuración de la instalación para el usuario

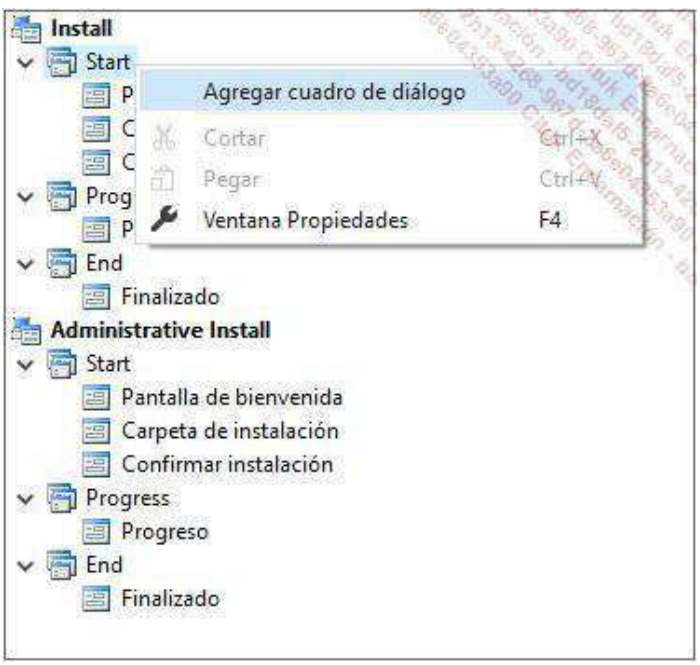
Los programas de instalación muestran generalmente una interfaz gráfica al usuario, de manera que pueda personalizar las acciones a realizar durante el proceso. Estas pantallas de configuración se pueden editar desde el menú **User interface**. La interfaz de trabajo muestra las diferentes pantallas en forma de árbol.



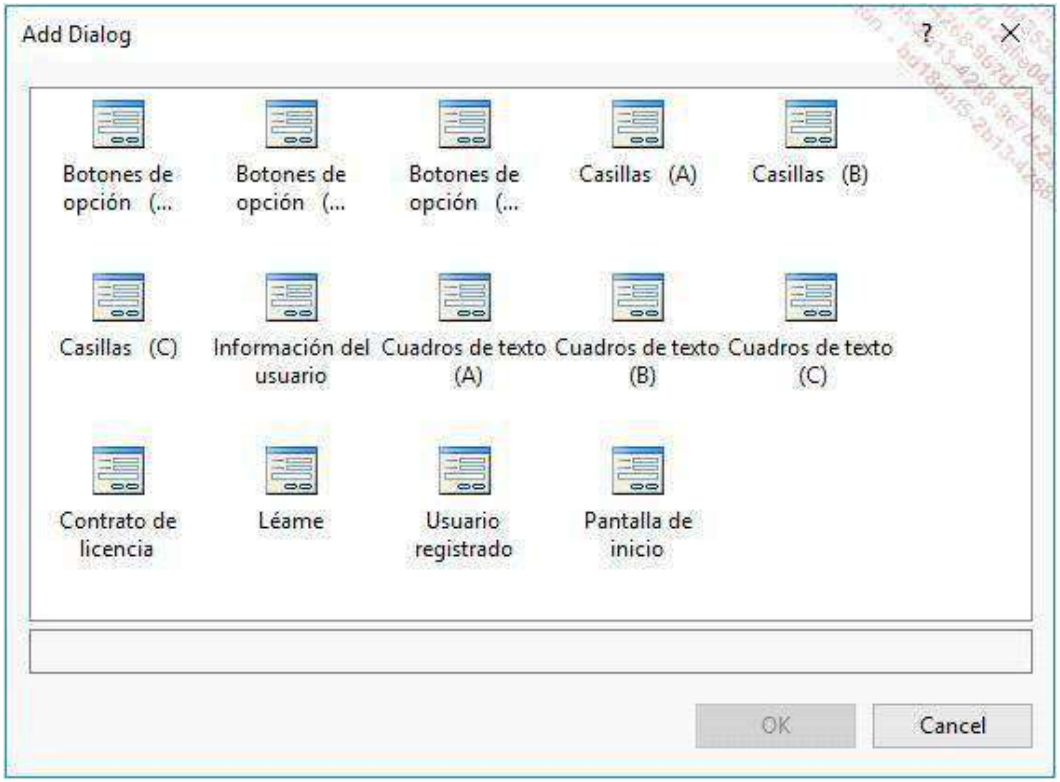
Las pantallas por defecto muestran diferentes propiedades que permiten modificar su contenido. En el caso de la pantalla de bienvenida, se llaman **BannerBitmap**, **CopyrightWarning** y **WelcomeText**. Permiten personalizar respectivamente la imagen mostrada en la parte superior, el texto de aviso sobre los derechos de autor y el mensaje de presentación de la instalación.



Para proporcionar una experiencia lo más completa posible, se puede agregar una pantalla personalizada al asistente de instalación. Haciendo clic con el botón derecho en una de las secciones de programa (**Start, Progress, End**) se abre un menú contextual con la opción **Agregar cuadro de diálogo**.

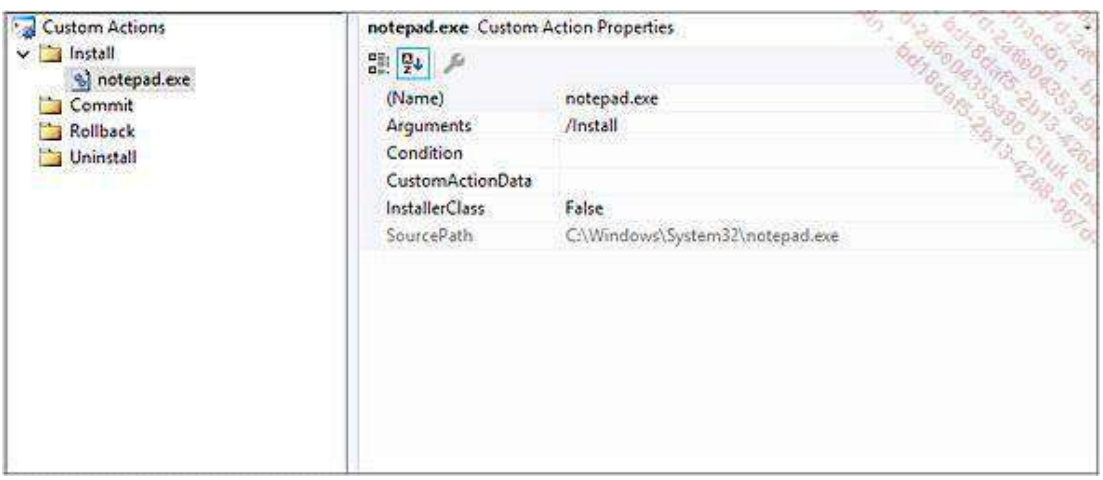


Al elegir esta opción vamos a una pantalla de selección que muestra los modelos de pantalla disponibles. Cada uno de estos modelos se puede personalizar asignando valores a sus propiedades.



d. Ejecución de acciones personalizadas

Las posiblidades que nos proporcionan las opciones ya comentadas a veces no cubren la totalidad de nuestras necesidades. Cuando se debe ejecutar una operación fuera de los casos predefinidos, el diseñador del proyecto de instalación puede definir una o varias acciones personalizadas. Estas corresponden sobre todo a la ejecución de archivos de script (.bat, .js, .vbs) pero también se pueden asociar a la ejecución de programas ejecutables.



Este menú también se puede utilizar para realizar operaciones fuera del ámbito de la instalación: abrir un archivo (o un sitio web) que presente la aplicación y sus posibilidades al finalizar la instalación o abrir un formulario web que permita al usuario opinar después de desinstalar.

e. Condiciones

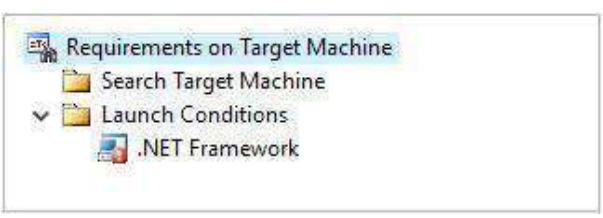
Todas las operaciones que afectan al sistema del usuario se pueden ejecutar de manera condicional. Esto puede ser útil si la aplicación necesita una dependencia (por ejemplo, Crystal Reports) para funcionar de manera óptima.

La dependencia se puede instalar si no se ha encontrado en el sistema del usuario.

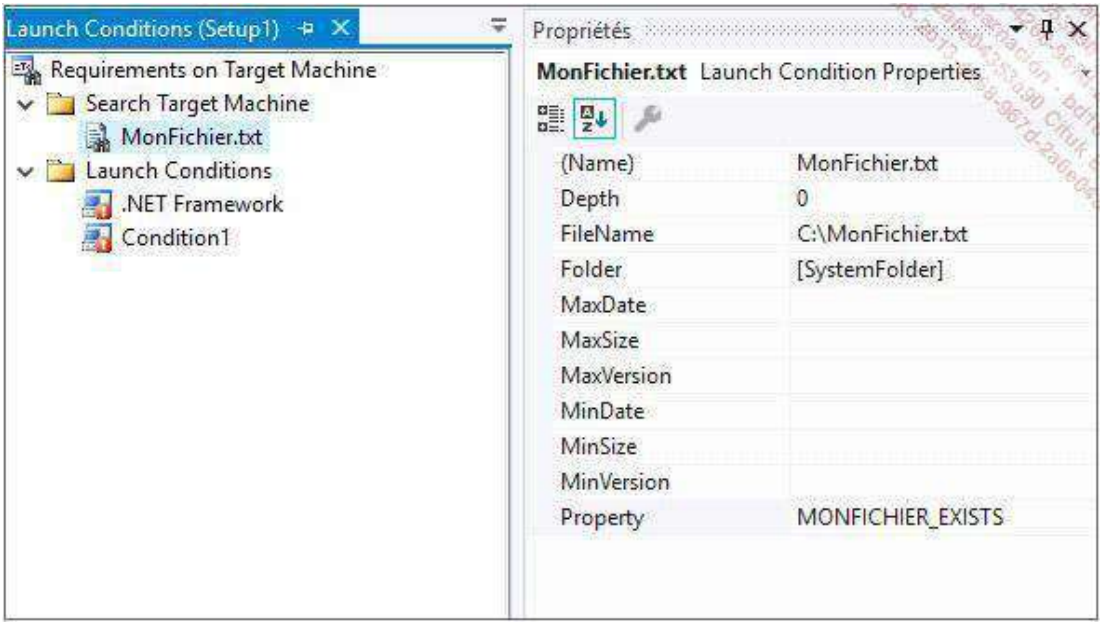
Los elementos que tienen la propiedad Condition se pueden configurar para adoptar un comportamiento en función del entorno de instalación. La sintaxis de definición de estas condiciones, así como todas las variables de entorno que se pueden evaluar, están fuera del ámbito de este libro. No obstante, la siguiente tabla enumera algunas de las condiciones más comunes que podrá utilizar.

Descripción	Condición
Primera instalación de la aplicación	NOT Installed
Desinstalación	REMOVE~="ALL"
Modificación de la instalación	Installed AND NOT REINSTALL AND NOT REMOVE~="ALL"
Sistema operativo 64 bits	VersionNT64
Sistema operativo 32 bits	NOT VersionNT64
Sistema = Windows XP	VersionNT = 501
Sistema = Windows XP (64 bits)	VersionNT = 502 AND MsiNTProductType = 1
Sistema = Windows Server 2003	VersionNT = 502 AND MsiNTProductType = 3
Sistema = Windows 7	VersionNT = 601 AND MsiNTProductType = 1
Sistema = Windows Server 2008 R2	VersionNT = 601 AND MsiNTProductType = 3
Sistema = Windows 8 / Server 2012	VersionNT = 602
Sistema = Windows 10 / Server 2016	VersionNT = 603
Mínimo 2 Gb de RAM	PhysicalMemory >= 2048
Sistema operativo español	SystemLanguageID = 3082

También se pueden configurar otras condiciones mediante el menú contextual del proyecto: las condiciones de ejecución. Permiten validar la posibilidad de instalar una aplicación desde el inicio de la ejecución del programa de instalación.



La primera parte del árbol permite definir variables de entorno personalizadas, calculadas a partir de un resultado de búsqueda de un fichero, de una clave de registro o de una aplicación instalada. Estas variables se puede evaluar en las condiciones para permitir o no la instalación de la aplicación.



Una vez finalizado este proceso de configuración, el proyecto de despliegue está listo para ser generado.

ClickOnce

ClickOnce es una tecnología de la que pueden sacar provecho los desarrolladores para desplegar sus aplicaciones y sus actualizaciones. Simplifica el proceso de instalación y requiere una mínima intervención por parte del usuario final.

1. La tecnología ClickOnce

La tecnología ClickOnce se denomina "inteligente", pues permite superar ciertos escollos que se encuentran, con frecuencia, durante la fase de despliegue.

En términos generales, para poder instalar una aplicación, es necesario disponer de un ejemplar del programa de instalación correspondiente sobre un dispositivo físico o en descarga. En ambos casos, el usuario posee la totalidad de los componentes de la aplicación, a menudo como un paquete completo, antes de realizar la instalación efectiva.

Además de este modo de instalación clásico, ClickOnce ofrece la posibilidad de realizar la instalación de manera remota a través de Internet o de un recurso compartido de red. Se requiere, únicamente, un ejecutable mínimo para comenzar el proceso. Este está disponible, generalmente, mediante un vínculo en una página web o una intranet y no ocupa más que unos pocos kilobytes.

La gestión de las dependencias compartidas por varias aplicaciones puede volverse problemática muy rápidamente, en particular debido a conflictos entre versiones. ClickOnce simplifica esta gestión aislando las aplicaciones de manera que sean autónomas y no puedan encontrarse frente a situaciones de conflicto.

La actualización de una aplicación desplegada de manera tradicional supone, en la mayoría de casos, tener que reinstalar cada uno de sus componentes. Con ClickOnce, la instalación de las actualizaciones se realiza mediante una instalación incremental: solamente se instalan aquellos componentes que requieren una actualización.

Las aplicaciones desplegadas mediante ClickOnce pueden, también, utilizarse en modo conectado, es decir, los archivos de la aplicación no están instalados sobre el equipo local, sino que se ejecutan desde su ubicación remota. Este modo de funcionamiento se utiliza, a menudo, en un contexto empresarial, cuando se requiere que la aplicación reciba constantes actualizaciones.

a. Principios de funcionamiento

El principio de funcionamiento de ClickOnce se basa en la existencia de dos archivos de descripción en formato XML llamados manifiestos:

- El manifiesto de la aplicación (archivo `.exe.manifest`),
- El manifiesto del despliegue (archivo `.application`).

El primero provee una descripción de la aplicación mediante sus ensamblados, sus dependencias y también autorizaciones administrativas necesarias para su ejecución. Expone, también, la localización de las actualizaciones que se pondrán a disposición de la aplicación.

El manifiesto de despliegue describe cómo se despliega la aplicación o qué versiones de framework .NET son compatibles con ella. También se registra en este archivo información relativa a la versión de la aplicación o el fabricante de software.

Tras la publicación, estos archivos, así como los componentes de la aplicación, se copian en una carpeta accesible a los usuarios en función del método de despliegue seleccionado. El manifiesto y los archivos de la aplicación no deben almacenarse, necesariamente, en el mismo lugar que el manifiesto de despliegue, pero sí deben estar accesibles para los usuarios y su ubicación debe estar correctamente referenciada en el manifiesto de despliegue.

Cuando se abre el manifiesto de la aplicación haciendo clic sobre un vínculo en una página web o haciendo doble clic en el explorador de Windows, se abre una ventana que solicita la confirmación de la acción al usuario. Cuando se confirma, ClickOnce se encarga de instalar o de abrir la aplicación, en función del modo de funcionamiento configurado.

b. Métodos de despliegue disponibles

ClickOnce proporciona tres estrategias para el despliegue de una aplicación. La opción seleccionada depende esencialmente del tipo de aplicación y del públic al que se dirija.

Instalación a partir de un CD-ROM

Esta estrategia se corresponde con la instalación de la aplicación desde un soporte extraíble, como un CD-ROM, un DVD o una llave USB. Cuando el usuario selecciona instalar la aplicación, se configura un acceso directo en su menú de **Inicio** y se agrega una entrada en la lista **Agregar o quitar programas** del **Panel de control**.

Esta estrategia está bien adaptada para clientes cuya conectividad de red sea limitada o inexistente, con más razón si el tamaño de la aplicación es importante. El uso de un soporte extraíble no requiere, en efecto, ninguna conectividad. No obstante, la búsqueda de actualizaciones tras el arranque de la aplicación depende de un acceso a la red disponible en la máquina.

Instalación desde la Web o una ubicación de red

Esta estrategia instala la aplicación en el disco duro del cliente agregando un acceso directo en el menú de **Inicio** así como una entrada en la lista **Agregar o quitar programas** del **Panel de control**, pero la instalación debe realizarse desde una ubicación web o un recurso compartido de red. En el primer caso, el usuario hace clic, por lo general, sobre un enlace en una página web, mientras que la instalación desde una ubicación de red se realiza, generalmente, haciendo doble clic sobre el manifiesto de despliegue en un recurso de red compartido.

Este tipo de instalación requiere conectividad de red suficiente para el tamaño de la aplicación y es ideal, por lo general, para desplegar una aplicación en la empresa. Se utiliza también, en ocasiones, cuando la aplicación debe distribuirse a un gran público muy disperso geográficamente.

Arranque desde un sitio web o una ubicación de red

Esta estrategia no despliega la aplicación de manera local en las máquinas cliente. Se comporta, de manera global, como una aplicación web, es decir, cuando el usuario hace clic sobre un enlace web, la aplicación se descarga y ejecuta, y tras su cierre ya no está disponible de manera local. Como no se ha realizado ninguna instalación, no se crea ningún acceso directo para la aplicación y tampoco aparece en la lista **Agregar o quitar programas**.

Una condición para utilizar esta estrategia es la existencia de una conexión de Internet adaptada en el lugar de trabajo de cada cliente. Este tipo de despliegue se utiliza cuando las aplicaciones se ejecutan con poca frecuencia, para evitar utilizar recursos de almacenamiento cuando la aplicación no se utiliza.

c. Actualización de aplicaciones con ClickOnce

El proceso de actualización puede automatizarse mediante el motor de ejecución de ClickOnce. La información vinculada a la estrategia de actualización se almacena en el manifiesto de despliegue en la etapa de publicación.

La aplicación instalada lee periódicamente la información registrada en este manifiesto de despliegue para verificar la disponibilidad de actualizaciones. Si existe una nueva versión disponible, ClickOnce descarga únicamente los

archivos necesarios para mantener la aplicación actualizada y los instala. Si el origen de las actualizaciones no está accesible o disponible, la aplicación se ejecuta sin realizar esta verificación.

Existen tres estrategias que permiten gestionar estas actualizaciones:

- La verificación automática durante el arranque de la aplicación.
- La verificación automática durante la ejecución de la aplicación, como tarea de fondo.
- La existencia de una interfaz gráfica destinada a gestionar las actualizaciones.

Es posible, también, configurar la frecuencia de búsqueda de actualizaciones o bien forzar una actualización obligatoria.

Verificación durante el arranque

Cuando el usuario arranca la aplicación, ClickOnce lee el manifiesto de despliegue y busca las actualizaciones disponibles para la aplicación. Si existe una nueva versión disponible, se descarga e instala automáticamente. Una vez terminada la instalación, la aplicación se ejecuta.

Verificación durante la ejecución

Por defecto, ClickOnce verifica la existencia de actualizaciones durante la ejecución de la aplicación. Esta búsqueda se realiza como tarea de fondo, de manera transparente al usuario. Si existe una nueva versión disponible, el usuario puede descargarla e instalarla. El siguiente arranque de la aplicación ejecutará la versión actualizada.

Intervalo de actualización

Es posible definir la frecuencia para la búsqueda de actualizaciones de una aplicación. El intervalo entre las verificaciones puede definirse en términos de horas, de días o de semanas. Si no existe ninguna conexión a la red disponible en el momento de realizar esta búsqueda, se pospone a la siguiente ejecución de la aplicación.

Actualizaciones obligatorias

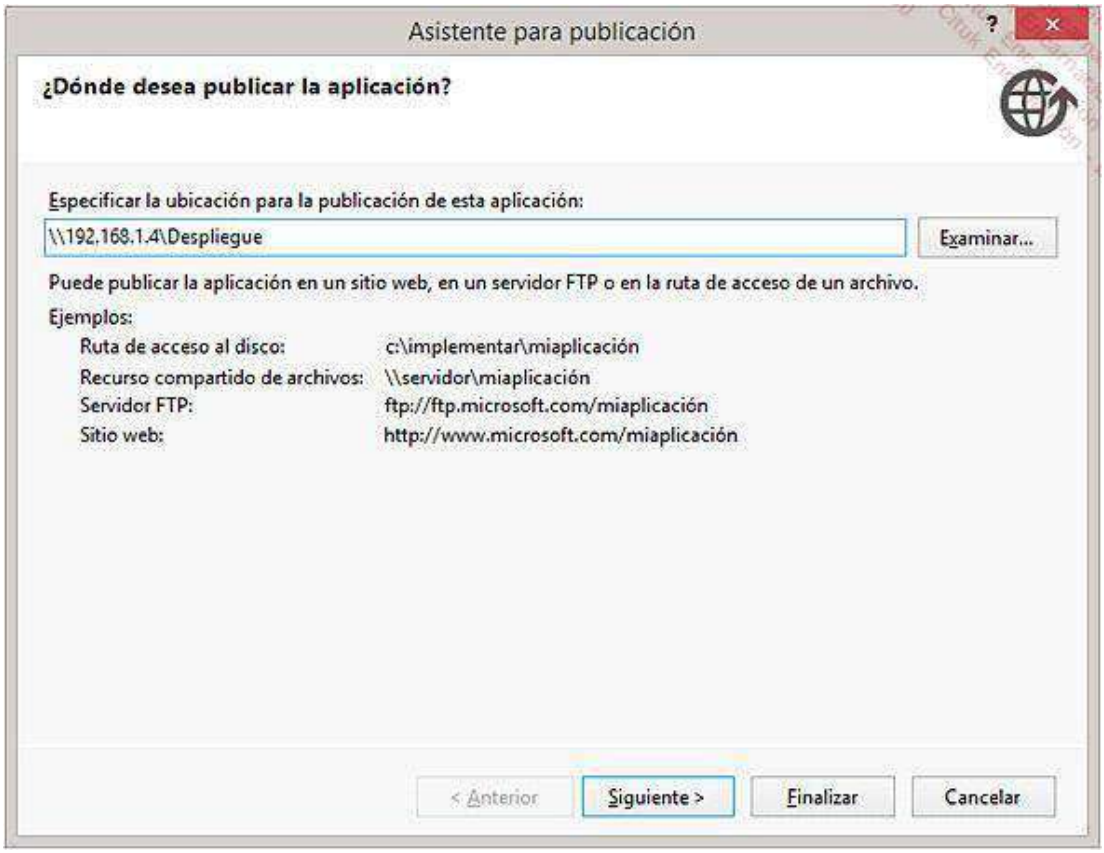
En ocasiones resulta necesario que los usuarios se vean forzados a utilizar la última versión de la aplicación, en particular en caso de que se produzca alguna modificación importante susceptible de producir problemas de funcionamiento en las antiguas versiones de la aplicación. En este caso es posible marcar la actualización como obligatoria, lo que impide la ejecución de versiones más antiguas de la aplicación.

2. La publicación ClickOnce

El despliegue de una aplicación con ClickOnce se realiza mediante un asistente en Visual Studio. Este asistente permite introducir la información necesaria a lo largo de una serie de páginas.

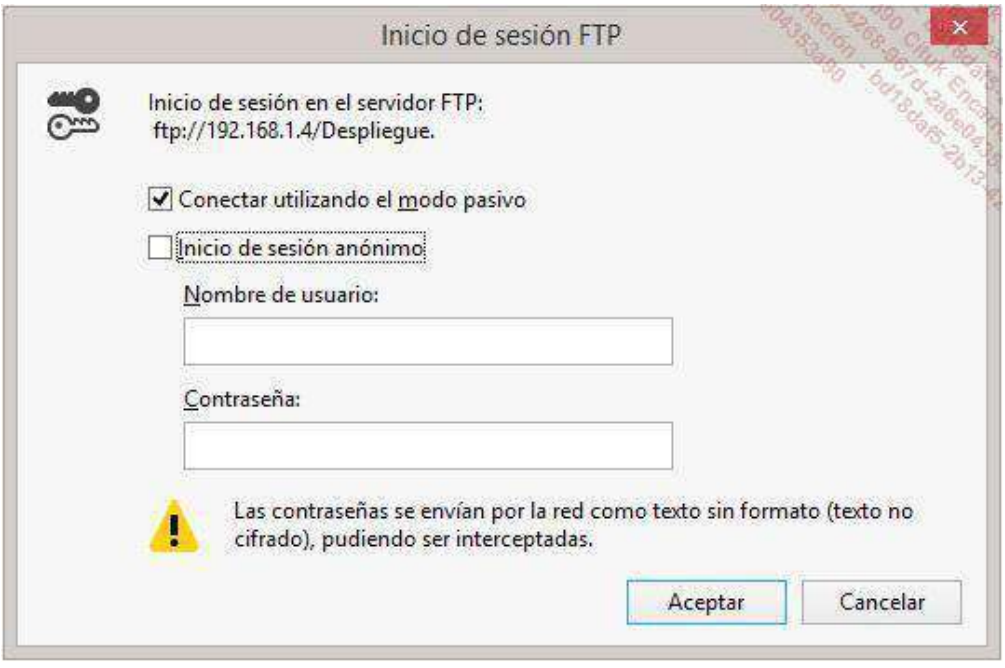
El arranque del asistente para publicación se realiza mediante la opción **Publicar**en el menú contextual del proyecto que se desea desplegar.

La primera etapa permite definir la ubicación en la que debe publicarse el proyecto.

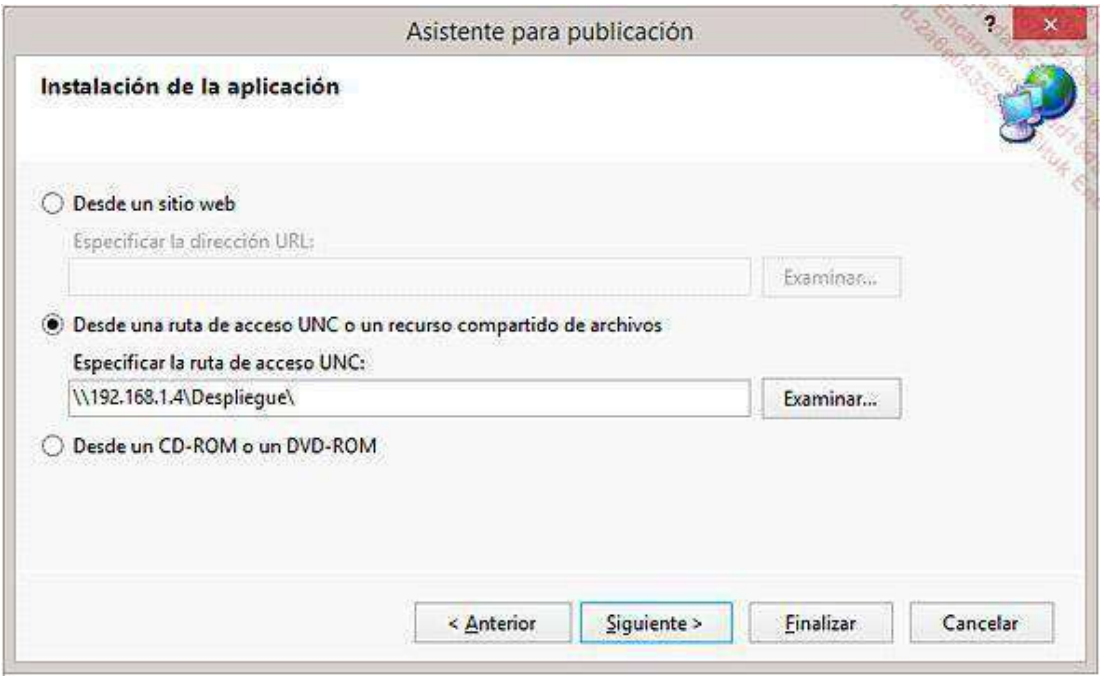


Esta ubicación puede ser:

- Una carpeta de la máquina local.
- Una carpeta compartida situada en otra máquina. Debe proveerse como una ruta UNC y el usuario en curso debe disponer de permisos de escritura sobre esta carpeta.
- Un servidor web IIS sobre el que se crea y configura una carpeta virtual para acoger los archivos.
- Un servidor FTP para el que es necesario proporcionar cierta información de conexión. Esta información puede introducirse en el siguiente cuadro de diálogo, que se abre tras la validación de la última etapa del asistente para publicación.

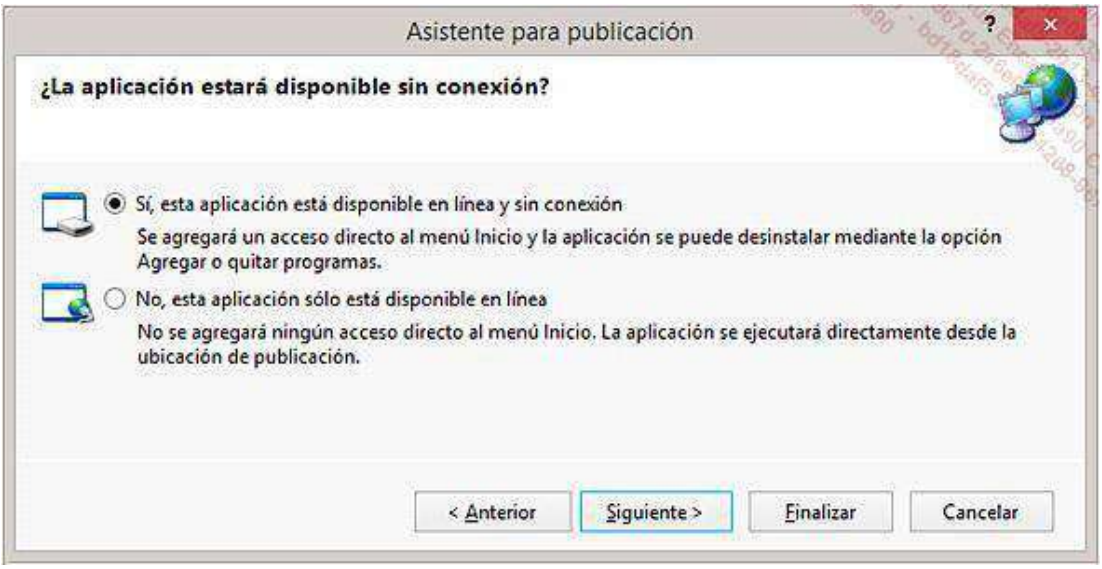


La siguiente etapa permite indicar el método de despliegue que debe utilizarse para la aplicación.

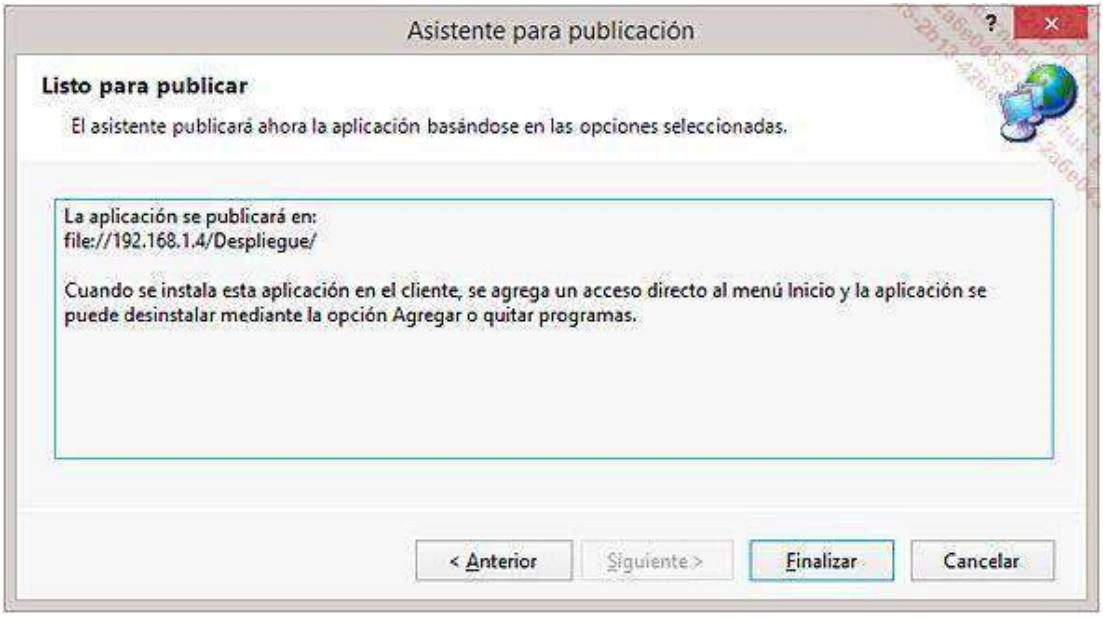


La selección de las opciones **Desde un sitio web** o **Desde una ruta de acceso UNC o un recurso compartido de archivos** requiere introducir una URL o una ruta UNC desde la que se instalará la aplicación.

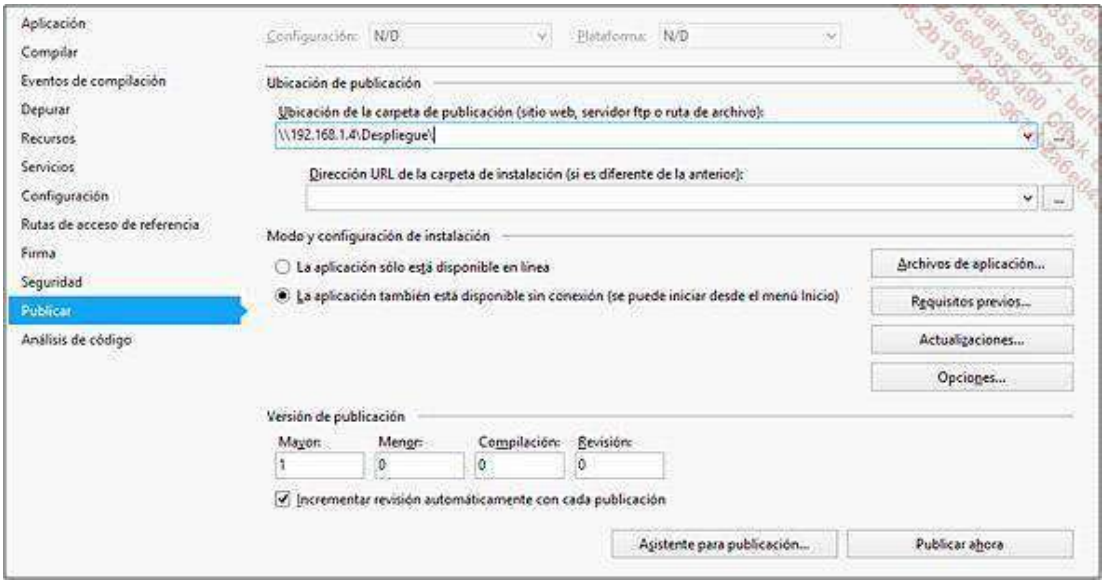
La siguiente pantalla permite seleccionar el modo de ejecución de la aplicación:



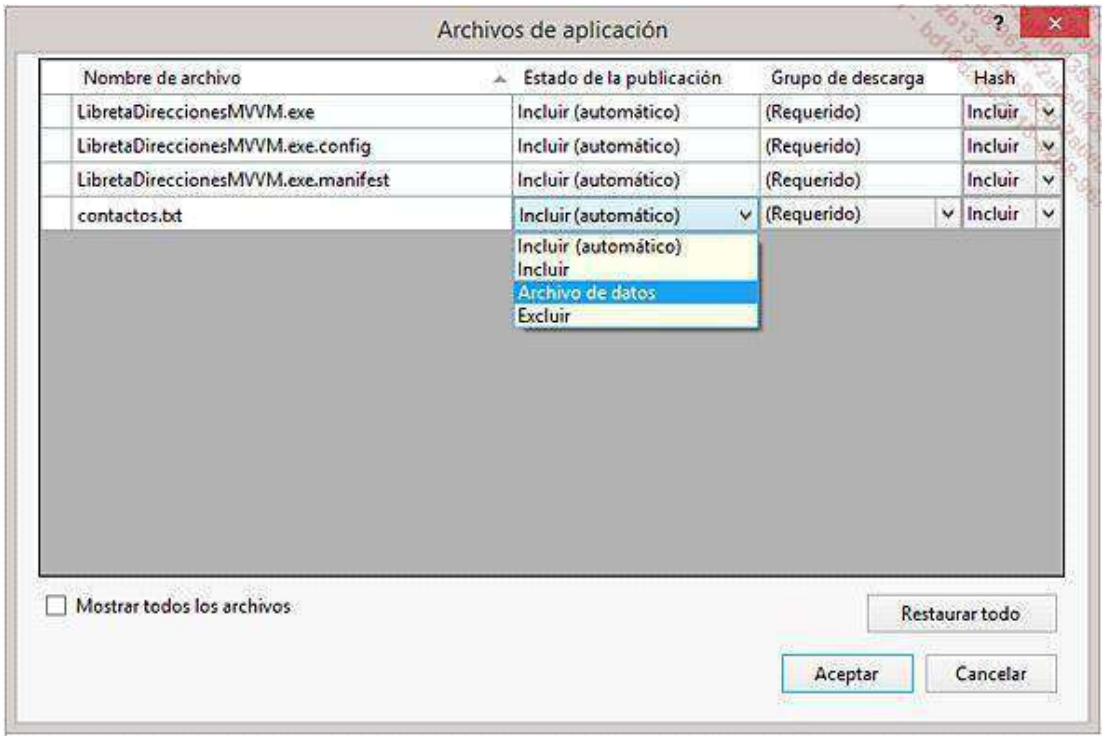
Tras la validación de esta página, se abre una nueva pantalla que recuerda la ubicación de destino de la publicación. Si se hace clic en el botón **Finalizar** se lanza la ejecución efectiva de la publicación.



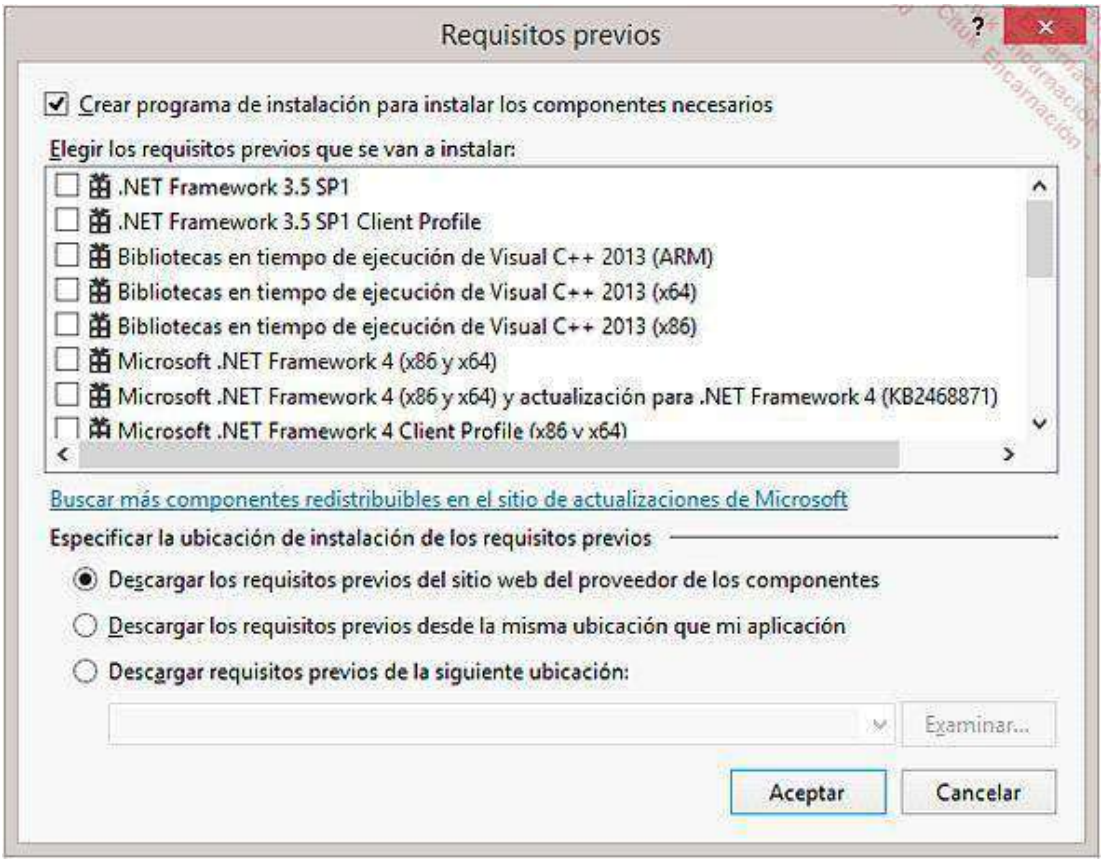
Existen otras opciones de publicación disponibles en la sección **Publicar** de las propiedades del proyecto.



El botón **Archivos de aplicación** abre una ventana que permite indicar los archivos a desplegar. Los archivos generados por la compilación del proyecto se incluyen automáticamente. Es posible incluir otros archivos como **Archivos de datos**.

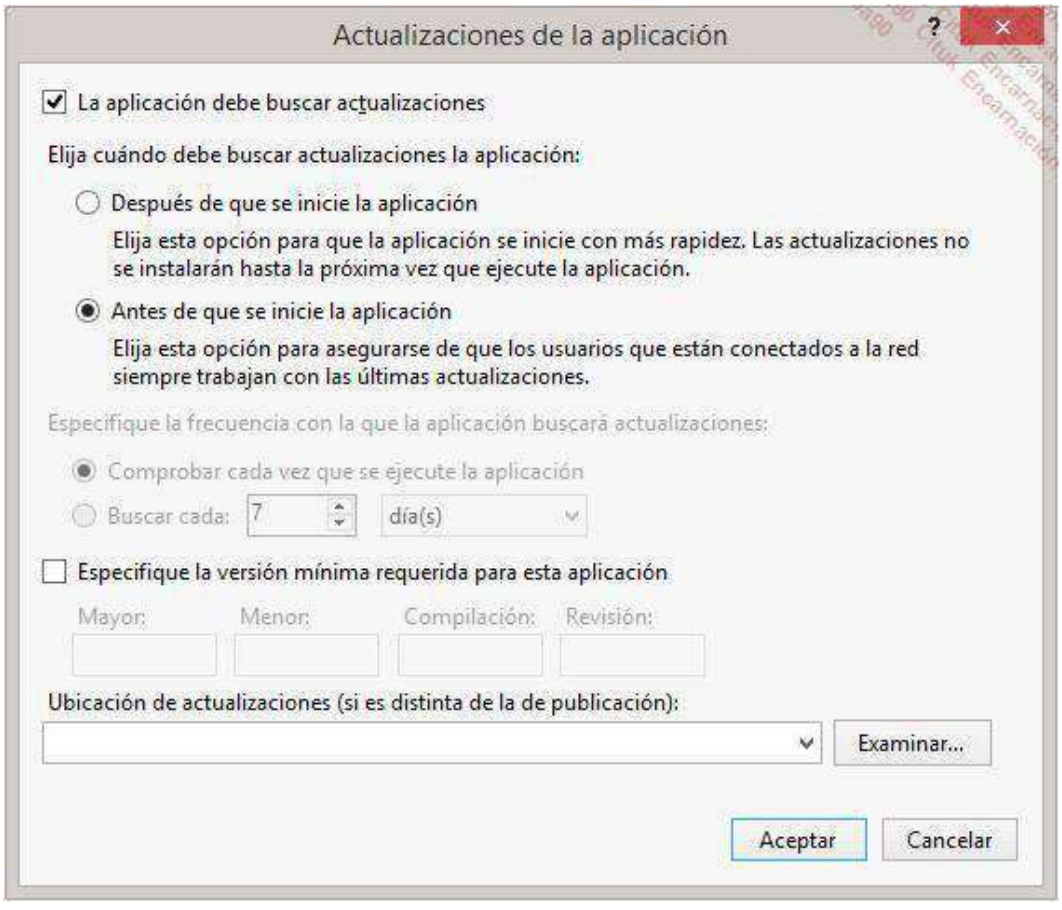


El botón **Requisitos previos** permite seleccionar ciertos componentes imprescindibles para el correcto funcionamiento de la aplicación.

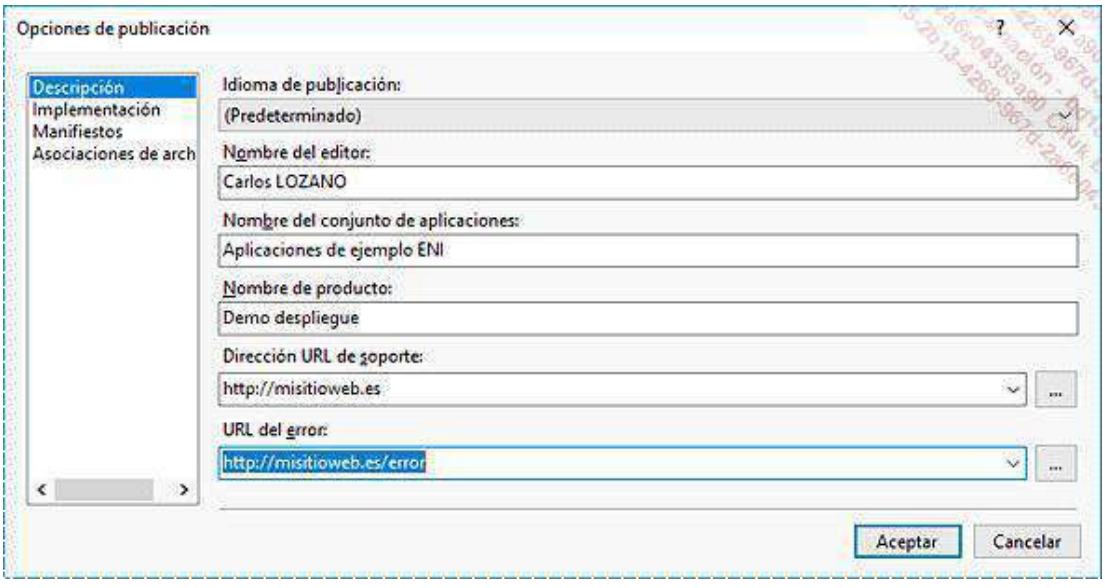


Es posible generar un programa de instalación para los componentes requeridos marcando la opción **Crear programa de instalación para instalar los componentes necesarios**. La fuente de instalación de los componentes requeridos debe precisarse mediante las opciones situadas en la parte inferior de esta ventana.

El botón **Actualizaciones** permite indicar ciertos parámetros relativos, en particular, a la frecuencia de verificación de las actualizaciones o a su modo de búsqueda.



El último botón, llamado **Opciones**, abre una ventana de configuración de distintos parámetros vinculados con el despliegue.



Idioma de publicación

Indica el idioma en el que se publicará la aplicación.

Nombre del editor

Especifica el nombre del desarrollador o de la compañía que edita la aplicación. Si este campo no se informa, se utiliza un valor por defecto.

Nombre del conjunto de aplicaciones

Indica el nombre de la suite de aplicaciones de la que forma parte la aplicación. Este nombre se utiliza para dar nombre a la carpeta en la que se instala la aplicación.

Nombre de producto

Este parámetro se corresponde con el nombre de la aplicación. Cuando este campo no tiene ningún valor, se utiliza el nombre del ensamblado para darle valor.

Dirección URL de soporte

Este parámetro indica el sitio web que puede proporcionar datos de asistencia para la aplicación.

Implementación de la página web

El valor de este parámetro se corresponde con el nombre de la página web de despliegue.

Utilizar la extensión de archivo ".deploy"

Los servidores web se configuran, a menudo, para bloquear la descarga de archivos cuya extensión no esté autorizada. Los archivos ejecutables y las bibliotecas de clases están, generalmente, filtrados. Por este motivo, el despliegue con ClickOnce puede cambiar el sufijo del nombre de cada uno de los archivos de la aplicación por la extensión `.deploy`. De este modo, basta con autorizar tres extensiones en la configuración del sitio web para que la aplicación pueda ejecutarse:

- `.application`
- `.manifest`
- `.deploy`

Bloquear la aplicación para que no se active mediante una dirección URL

Cuando se marca esta opción, la aplicación no puede arrancar desde la URL asociada a su manifiesto de despliegue. Es obligatorio utilizar el menú de **Inicio**de la máquina.

Permitir que se pasen los parámetros de la dirección URL a la aplicación

Si esta opción está marcada, la aplicación puede acceder a los parámetros de la URL utilizada para ejecutar la aplicación. Puede, también, tratarlos y proporcionar un comportamiento particular en función de estos parámetros.

Usar el manifiesto de la aplicación para la información de confianza

La selección de esta opción le permite firmar de nuevo el manifiesto de la aplicación mediante un certificado que contenga la información que desee.

Introducción-Glosario

A continuación encontrará un resumen de las palabras clave de C# que se abordan a lo largo de este libro, ordenadas alfabéticamente. Cada una se acompaña de una breve descripción y de un ejemplo de uso sencillo.

abstract

Si se utiliza con una clase, define que la clase no pueda instanciarse.

Si se utiliza con una función/propiedad, indica que su implementación deberá proveerse en un tipo derivado.

```
public abstract class Automóvil
{
    public abstract void Arrancar();
}
```

as

Operador de conversión de tipo. Cuando una variable no puede convertirse al nuevo tipo, devuelve un valor `null`.

```
string miCadena = miObjeto as string;
```

ascending

Define una ordenación ascendente cuando se utiliza con la palabra clave `orderby` en una consulta LINQ.

```
var consulta = from c in ListaClientes
               orderby c.Apellidos ascending
               select c;
```

async y await

`async` define que un método pueda ejecutarse de manera asíncrona. El tipo de retorno de los métodos asíncronos debe ser `void`, `Task` o `Task<T>`.

`await` permite esperar a que termine la ejecución de una tarea asíncrona antes de ejecutar el resto del código del método.

```
public async Task<string> EjecutarAsync()
{
    return await Task.Run(() => "¡Hola!");
}
```

base

Proporciona acceso a un miembro de la clase base.

```
public class MiTipo : Object
{
    public string Mostrar()
    {
            string texto = base.ToString();
            Console.WriteLine(texto);
    }
}
```

bool

Alias C# del tipo de datos `System.Boolean`. Una variable de este tipo puede tener los valores `true` o `false`.

```
bool seMuestra = true;
```

break

En un bloque `switch`, debe estar presente al final de cada etiqueta `case` en la que no se incluye ninguna instrucción `return`.

```
switch (valor)
{
    case 0:
          Console.WriteLine("Caso 0");
          break;
}
```

En un bucle, permite salir del bucle.

```
int i = 0;
while (true)
{
    if (i == 7)
          break;
    i++
}
```

byte

Alias C# del tipo `System.Byte`. Una variable de este tipo contiene un valor numérico entero comprendido entre 0 y 255.

```
byte number = 42;
```

case

Asocia un bloque de instrucciones con un valor numérico o de enumeración en un bloque `switch`.

```
switch (valor)
{
    case 0:
          Console.WriteLine("Caso 0");
          break;
}
```

catch

Define un bloque de gestión de excepciones a continuación de un bloque `try`.

```
try
{
    System.IO.File.Open("C:\\archivo.txt",
System.IO.FileMode.Open);
}
catch (System.IO.FileNotFoundException)
{
   //Gestión de las excepciones de tipo FileNotFoundException
}
catch (Exception)
{
   //Gestión de todas las excepciones no manejadas
}
```

char

Alias C# del tipo `System.Char`. Contiene un carácter Unicode definido por su forma textual o numérica.

```
char letraC = 'C';
char letraCNum = '\u0063';
```

class

Define una clase.

```
class MiTipo
{
}
```

const

Marca una variable como constante. Las constantes son tipos concretos de variables estáticas, accesibles mediante el nombre del tipo al que pertenecen.

```
public const string MiArchivoDeConfiguración = "config.xml";
```

continue

Permite interrumpir la ejecución de la iteración de un bucle para pasar a la siguiente iteración.

```
for (int i = 0; i < 100; i++)
{
    if (i % 2 == 0)
        continue;
}
```

decimal

Alias C# del tipo `System.Decimal`. Una variable de este tipo contiene un valor numérico decimal comprendido entre ±1,0e-28 y ±7,9e28.

```
decimal valorDecimal = 42.000042;
```

default

Se utiliza como etiqueta por defecto en un bloque `switch` y permite ejecutar una sección de código si no existe ninguna otra etiqueta que se corresponda con el valor comprobado.

```
switch (valor)
{
    case 0:
          Console.WriteLine("Caso 0");
          break;
    case 1:
          Console.WriteLine("Caso 1");
          break;
    default:
          Console.WriteLine("Ni el caso 0, ni el caso 1");
          break;

}
```

También se puede utilizar para obtener el valor por defecto para un tipo particular.

```
int valorDefectoInt = default(int);
```

delegate

Tipo que representa una firma de función. Los delegados se utilizan, en particular, para fijar la firma de los controladores de eventos.

```
public delegate void EventHandler(object sender, EventArgs args);
```

descending

Define una ordenación descendente cuando se utiliza con la palabra clave `orderby` en una consulta LINQ.

```
var consulta = from c in ListaClientes
               orderby c.Apellidos descending
               select c;
```

do

Se utiliza para comenzar una estructura de bucle `do ... while`. Este tipo de bucle se ejecuta como mínimo una vez antes de evaluar la condición de salida.

```
do
{
   Console.WriteLine(i);
} while (i < 5);
```

double

Alias C# del tipo `System.Double`. Una variable de este tipo contiene un valor numérico decimal comprendido entre ±5,0e-324 y ±1,7e308.

```
double valorDecimal = 42.000042;
```

dynamic

Permite utilizar variables cuyo tipo es desconocido en el momento de la ejecución.

```
dynamic variableDynamic = "un valor";
```

else

Se utiliza en la estructura de evaluación de una condición `if ... else if ... else`. Define un bloque que gestiona los casos en los que no se cumple(n) la(s) condición(es) anterior(es).

```
if (valor == 5)
{
}
else if (valor > 5)
{
}
else
{
}
```

enum

Define una enumeración.

```
public enum TipoPersona
{
    Contacto,
    Cliente
}
```

event

Define un evento que puede generarse mediante su clase padre.

```
public event EventHandler EjecuciónTerminada;
```

false

Uno de los dos valores para el tipo `System.Boolean`.

```
bool estáActivo = false;
```

finally

Se utiliza en la escritura de estructuras `try ... catch ... finally`. Define un bloque de código que se ejecuta incluso aunque el bloque `try` genere una excepción.

```
try
{
}
finally
```

```
{
}
```

float

Alias C# del tipo `System.Single`. Una variable de este tipo contiene un valor numérico decimal comprendido entre ±1,5e-45 y ±3,4e38.

```
float valorDecimal = 42.000042;
```

for

Define un bucle para el que se especifica una instrucción de inicialización, una condición de salida y una instrucción de iteración. Su número de iteraciones se conoce, generalmente, antes de su ejecución.

```
for (int i = 0; i < 5; i++)
{
}
```

foreach

Define un bucle que recorre todos los elementos de una colección que implementa la interfaz `IEnumerable` o su versión genérica `IEnumerable<T>`.

```
string[] array = new string[10];
foreach (string elemento in  array)
{
}
```

from

Define un origen de datos utilizado en una consulta LINQ.

```
var consulta = from c in ListaClientes
               select c;
```

get

Define un bloque de código que se ejecuta cuando se lee una propiedad. Si este bloque no está definido, la propiedad es de solo escritura.

```
public string Mensaje
{
```

```
    get { return "Un mensaje";  }
}
```

goto

Define un salto en la ejecución del código hasta una etiqueta.

```
Console.WriteLine("Antes del goto");
goto miEtiqueta;
Console.WriteLine("Tras el goto: nunca se ejecuta");
miEtiqueta:
Console.WriteLine("Tras la etiqueta");
```

if

Define un bloque de código que se ejecuta únicamente si la evaluación de la condición asociada devuelve el valor `true`.

```
if (i == 5)
{
}
```

in

Se utiliza en la definición de una estructura `foreach` o en una consulta LINQ para indicar la colección de origen de una variable.

```
foreach (string elemento in array)
{
}
```

```
var consulta = from cliente in context.Clientes
               select cliente.Apellidos;
```

int

Alias C# del tipo `System.Int32`. Una variable de este tipo contiene un valor numérico entero comprendido entre -2 147 483 648 y 2 147 483 647.

```
int númeroPájaros = 123;
```

interface

Define una interfaz. El nombre de una interfaz empieza, por convención, por una letra i mayúscula.

```
public interface ITestable
{
}
```

internal

Modificador de visibilidad que permite hacer un elemento accesible para todos los tipos definidos en el mismo ensamblado.

```
internal class ClaseInterna
{
}
```

is

Operador de comparación de tipo. Devuelve `true` si la variable indicada es del tipo especificado.

```
object valor = "Esto es una cadena de caracteres";
bool esUnaCadena = valor is string;
```

También se utiliza en el ámbito de la coincidencia de patrones (pattern matching).

```
objeto o = 10;
if (o is int i) Console.WriteLine(i);
```

join

Permite realizar una unión entre dos colecciones de datos en una consulta LINQ.

```
var consulta =
    from c in ListaClientes
    join p in ListaPedidos on c.IdCliente = p.IdCliente
    select new { c.IdCliente, p.IdPedido };
```

long

Alias C# del tipo `System.Int64`. Una variable de este tipo contiene un valor numérico entero comprendido entre -9 223 372 036 854 775 808 y 9 223 372 036 854 775 807.

```
long númeroPájaros = 123;
```

namespace

Define un espacio de nombres.

```
namespace MiAplicación.ViewModel
{
}
```

new

Operador de instanciación.

```
Cliente cliente = new Cliente();
```

También puede utilizarse para definir una restricción sobre un elemento genérico. En este contexto, fuerza a un tipo parámetro a que tenga un constructor sin parámetro.

```
public class Lista<T> where T : new ()
{
}
```

null

Valor especial que indica que un objeto de tipo referencia no posee ningún valor.

```
Cliente cliente = null;
if (cliente == null)
    cliente = new Cliente();
```

object

Alias C# del tipo `System.Object`. Es la clase base de todos los tipos de .NET.

```
object obj = "Cadena de caracteres registrada en una variable
de tipo object";
```

operator

Permite redefinir un operador estándar de C#.

```
public static ColecciónDeSellos operator +(ColecciónDeSellos
```

```
colección1, ColecciónDeSellos colección2)
{
   ColecciónDeSellos resultado;
   resultado.númeroDeSellos = colección1.númeroDeSellos +
colección2.númeroDeSellos;
   resultado.precioEstimado = colección1.precioEstimado +
colección2.precioEstimado;

   return resultado;
}
```

orderby

Define que los resultados de una consulta LINQ deben ordenarse en función de una o varias columnas.

```
var consulta = from c in ListaClientes
               orderby c.Apellidos descending
               select c;
```

out

Define que el parámetro de un método está disponible en escritura.

```
private void ParámetroOut(out in parámetro1, out in párametro2)
{
    parámetro1 = 5;
    parámetro2 = 42;
}

ParámetroOut (out var param, out);
```

override

En una clase derivada, indica que un método se ha redefinido.

```
public override string ToString()
{
    return "¡Hola!";
}
```

params

Indica que el parámetro de un método puede contener un número variable de elementos.

```
static void Main(params string[] args)
{
```

```
}
```

partial

Define que una clase o un método pueden tener varias partes.

```
public partial class ClaseParcial
{
}
```

private

Modificador de visibilidad que permite hacer que un elemento esté visible únicamente para aquel que lo contiene.

```
private string cadena = "valor";
```

protected

Modificador de visibilidad que permite hacer que un elemento esté visible únicamente para la clase que lo contiene así como sus clases derivadas.

```
protected string cadena = "valor";
```

public

Modificador de visibilidad que permite hacer que un elemento esté visible para todos los tipos.

```
public string cadena = "valor";
```

ref

Define el parámetro de un método como disponible en lectura y escritura.

```
private void ParámetroRef(ref int parámetro)
{
    parámetro = 42;
}
```

return

En un procedimiento, fuerza el retorno al método que lo invoca. En una función, fuerza el retorno al método que la

invoca devolviendo un valor.

```
public void Procedimiento()
{
    return;
}
public int Función()
{
    return 42;
}
```

sbyte

Alias C# del tipo `System.SByte`. Una variable de este tipo contiene un valor numérico entero comprendido entre -128 y 127.

```
sbyte númeroPájaros = 123;
```

sealed

Indica que una clase no puede utilizarse como clase de base en una relación de herencia.

```
public sealed class ClaseSellada
{
}
```

select

Indica qué resultado debe devolver una consulta LINQ.

```
var consulta = from c in ListaClientes
               select c;
```

set

Define un bloque de código que se ejecuta cuando se escribe una propiedad. Si el bloque no está definido, la propiedad es de solo lectura.

```
public string Mensaje
{
     set { this.mensaje = value; }
}
```

short

Alias C# del tipo `System.Int16`. Una variable de este tipo contiene un valor numérico entero comprendido entre -32.768 y 32.767.

```
short númeroPájaros = 123;
```

static

Indica que un miembro de la clase está vinculado con el tipo y no con una instancia concreta del tipo. Los miembros estáticos se utilizan con la sintaxis `NombreDeLaClase.Miembro`.

El uso de la palabra clave `static` con una clase indica que esta no es instanciable y que posee únicamente miembros estáticos.

```
public static class ClaseEstática
{
    public static void MétodoEstático
    {
    }
}
```

string

Alias C# del tipo `System.String`. Una variable de este tipo contiene una cadena de caracteres.

```
string nombreCliente = "Antonio";
```

struct

Define una estructura.

```
public struct MiEstructura
{
}
```

switch

Ejecuta una sección de código en función de un valor numérico o de una enumeración. Cada uno de los casos tratados se define mediante un bloque case.

```
switch (valor)
{
    case 0:
          Console.WriteLine("Caso 0");
```

```
            break;
    case 1:
            Console.WriteLine("Caso 1");
            break;
    default:
            Console.WriteLine("Ni el caso 0, ni el caso 1");
            break;

}
```

this

Hace referencia a la instancia actual de la clase que contiene el código.

```
var objetoActual = this;
```

throw

Permite generar una excepción.

```
throw new Exception("¡Esto es una excepción!");
```

Si se utiliza solo en un bloque `catch`, vuelve a lanzar la excepción en curso.

```
catch (Exception ex)
{
    throw;
}
```

true

Uno de los dos valores para el tipo `System.Boolean`.

```
bool estáActivo = true;
```

try

Define un bloque de instrucciones que pueden generar una excepción. Un bloque `try` debe estar seguido, obligatoriamente, de un bloque `catch` y/o `finally`.

```
try
{
}
catch (Exception ex)
```

```
{
}
finally
{
}
```

uint

Alias C# del tipo `System.UInt64`. Una variable de este tipo contiene un valor numérico entero sin signo comprendido entre 0 y 4 294 967 295.

```
uint númeroPájaros = 123;
```

ulong

Alias C# del tipo `System.UInt64`. Una variable de este tipo contiene un valor numérico entero sin signo comprendido entre 0 y 18 446 744 073 709 551 615.

```
ulong númeroPájaros = 123;
```

ushort

Alias C# del tipo `System.UInt16`. Una variable de este tipo contiene un valor numérico entero sin signo comprendido entre 0 y 65 535.

```
ushort númeroPájaros = 123;
```

using

Como encabezado de un archivo, importa un espacio de nombres en el archivo en curso.

```
using System.Data.SqlClient;
```

Utilizado en el código, permite definir un bloque que limpia automáticamente los recursos de un objeto que implementa la interfaz `IDisposable` cuando termina su uso.

```
using (SqlConnection conexión = new
SqlConnection("Server=localhost\\SQLEXPRESS;Initial
Catalog=Northwind;Integrated Security=True"))
{
}
```

value

Se utiliza únicamente en el bloque `set` de una propiedad. Se corresponde con el valor asignado a la propiedad.

```
public string Mensaje
{
     get { return this.mensaje; }
     set { this.mensaje = value;  }
}
```

var

Indica al compilador que debe determinar, él mismo, el tipo de la variable. Preste atención, no debe confundirse con `Dynamic`.

```
var cadena = "¡Hola!";
```

virtual

Marca un método, una propiedad o un evento como que puede redefinirse en una clase derivada de la clase en curso.

```
public virtual void MétodoVirtual()
{
}
```

void

Se utiliza como tipo de retorno para un procedimiento.

```
public void MiProcedimiento()
{
}
```

when

Se utiliza para crear filtros de excepciones. En una consulta LINQ, permite filtrar un juego de datos.

```
Try
{
    //...
}
catch (Exception ex) when (e.Message.Contains(" par"))
{
}
catch (Exception ex) when (e.Message.Contains("impar"))
```

```
{
}
```

where

En una consulta LINQ, permite filtrar un juego de datos.

```
var consulta = from c in ListaClientes
               where c.País equals "España"
               select c;
```

Permite, también, definir una restricción sobre un elemento genérico.

```
public class Lista<T> where T : IComparable
{
}
```

while

Define un bucle cuyo número de iteraciones depende del valor devuelto por la evaluación de una condición.

```
while (i < 50)
{
}
```