Vista nuestro Facebook:

https://www.facebook.com/dogr4mcode.web

Historia de la POO

La siguiente exposición no pretende ser exhaustiva. Sencillamente enumera los eventos que han conducido a la democratización de la programación orientada a objetos y a la creación del lenguaje C#.

Al contrario de lo que se podría pensar, el concepto de programación orientada a objetos no es reciente. En los años 60, dos brillantes investigadores noruegos, Quisten Negar y Ole-Johan Del, desarrollaron la base de la programación orientada a objetos, creando el lenguaje Simula. Las nociones básicas de la POO como las clases, la herencia, los métodos virtuales, etc., fueron creados en este lenguaje para permitir modelizar de manera fidedigna procesos industriales complejos. Simula-67 había abierto la vía de los lenguajes orientados a objetos, como Smalltalk y más adelante C++, Java y C# entre otros, que explotarían estos conceptos algunas décadas más tarde. Los dos autores de Simula fueron recompensados por sus trabajos a comienzos de los años 2000, es decir, justo antes de su desaparición.

Propuesto en 1980, Smalltalk es el primer lenguaje orientado a objetos disponible con un entorno de desarrollo gráfico integrado. Smalltalk fue diseñado por el equipo del americano Alan Kay, del centro de investigación informática californiana de XEROX (el famoso Palo Alto Research Center). Este lenguaje va a retomar y completar los conceptos básicos, fundamentalmente con la noción de compilación dinámica de un código "intermedio", portable entre entornos heterogéneos, en un código máquina destino. Java retomará este concepto con su Just In Time Compiler, y también C#.

Todavía en 1980, el danés Bjarne Stroustrup desarrolla para AT&T el lenguaje C++ (llamado originalmente C with classes). C++ es una evolución del lenguaje C inventado en UNIX por el célebre tándem canadiense-americano Kernighan y Ritchie, diez años antes. El lenguaje C++ todavía es muy utilizado, incluso por sus detractores, que lo juzgan como demasiado complejo y no muy orientado a objetos, gracias a su compatibilidad con C.

En los años 1990, James Gosling desarrolla el lenguaje Oak para Sun Microsystems. Oak se renombrará en Java en 1995. Este lenguaje fue pensado para ser independiente del hardware que ejecuta sus programas "precompilados", para ser robusto y seguro y, sobre todo, más sencillo de programar que C++. En la actualidad, el lenguaje Java se utiliza mucho tanto en entornos Web como en estaciones de trabajo, teléfonos y otros dispositivos, como las tabletas. Pertenece a Oracle.

En 2001, Microsoft presenta C# (pronunciado C Sharp - sharp es la traducción inglesa del sostenido en música y "C", la del do); lenguaje desarrollado por el danés Anders Hejlsberg, que no es otro que el creador de Turbo Pascal y el arquitecto de Delphi. C# es un lenguaje orientado a objetos cercano a Java, que permite programar aplicaciones en entornos Microsoft, como estaciones de trabajo, aplicaciones web, smartphones o tabletas.

¿Por qué programar con orientación a objetos?

En los años 70/80 aparecieron lenguajes procedurales con una ejecución lineal de sus programas, desde el principio hasta el final.

Reservado evidentemente a pequeños desarrollos, este tipo de programación evolucionó rápidamente hacia una programación llamada estructurada, donde procedimientos y funciones descomponían el programa en operaciones unitarias.

De manera similar al funcionamiento interno de los ordenadores, la programación estructurada no mezcla sencillamente variables y operaciones. Normalmente, en la parte superior del código hay una lista con la definición de las variables que se utilizarán a continuación en las diferentes funciones.

Las aplicaciones son cada vez más complejas y los entornos de ejecución evolucionan hacia interfaces gráficas, por lo que la programación estructurada rápidamente se hace difícil de codificar, mantener y evolucionar.

Inspirándose en los conceptos propuestos por Simula, llegan los lenguajes orientados a objetos.

La programación orientada a objetos aporta a los desarrolladores los medios de enfrentarse a sus nuevos retos, con:

- Una organización modular muy cercana a la realidad.
- Procesos de creación, puesta a punto y mantenimiento de los componentes, más sencillos y rápidos.
- La reutilización y evolución de los componentes existentes o de aquellos que provienen de proveedores de software de terceros.
- Una integración sencilla para su funcionamiento en entornos gráficos.
- Una lógica de codificación compatible con las aplicaciones distribuidas, que reparten sus contenidos en varias máquinas.
- Un desacoplamiento de la aplicación, que permite un trabajo en equipo más eficaz y productivo.

Historia de C#

Desde las primeras versiones de Windows, los lenguajes C y C++ se utilizaron mucho para construir aplicaciones. A pesar de su orientación a objetos y su potencia incontestable, C++ es un lenguaje complejo de utilizar. El desarrollador debe gestionar absolutamente todo, por ejemplo las asignaciones/desasignaciones de memoria y los aspectos relacionados con la gestión de la seguridad. Además, una ejecución directa de las aplicaciones en las interfaces Windows impacta inmediatamente en la estabilidad del sistema, en caso de funcionamientos incorrectos y bloqueos.

En los años 90, el lenguaje Java propone a los desarrolladores una codificación de sus programas mucho más sencilla, así como una ejecución aislada en una máquina virtual. Basta con que el entorno de explotación disponga de una máquina virtual Java para que la aplicación funcione sin modificación de código. Además, la máquina virtual garantiza un uso correcto de los recursos reales y asignados a la ejecución de las aplicaciones. Del lado de la gestión de la memoria, el desarrollador gestiona solo las peticiones de asignación, en función de las necesidades de su programa. Las zonas de memoria se explotan hasta el final de las operaciones y existe un dispositivo interno llamado recolector de basura, o garbage collector, que detecta las zonas de memoria que ya no se utilizan más y gestiona su liberación. Este funcionamiento controlado se llama ejecución "gestionada".

Microsoft, deseando proponer un entorno de desarrollo y ejecución gestionados al mismo tiempo para aplicaciones "clásicas" y para aplicaciones de Internet, presenta su .NET Framework 1.0 en 2002.

.NET y su CLR (Common Language Runtime) permiten la ejecución de aplicaciones con un código "intermedio" (MSIL) generado para cualquier tipo de compilador adaptado.

Entre un número impresionante de lenguajes "compatibles" MSIL, el lenguaje C# aparece como el mejor adaptado para explotar el framework .NET. Su sintaxis es muy parecida a la de Java, aunque existen diferencias entre ambos lenguajes que se describen perfectamente en los libros especializados. También es vecino de C y C++, animando al conjunto de desarrolladores Windows a dar el paso. La gestión de la memoria se simplifica gracias al recolector de basura de la CLR. Como veremos más adelante, "todo es un objeto" en C#. De hecho, el lenguaje está fuertemente tipado en comparación con C++, y esto limita los riesgos de error que se producen durante las conversiones demasiado permisivas de este último.

Enfoque procedural y descomposición funcional

Antes de enunciar los aspectos básicos de la programación orientada a objetos, vamos a repasar el enfoque procedural con ayuda de un ejemplo concreto de organización de código.

La programación procedural es un paradigma de programación que considera a los diferentes actores de un sistema como objetos prácticamente pasivos que un procedimiento central utilizará para una función dada.

Tomemos como ejemplo la distribución de agua corriente en los domicilios e intentemos emular este concepto en una aplicación muy sencilla. El análisis procedural (como el análisis de objetos) revela la siguiente lista de objetos:

- el grifo del lavabo;
- el depósito del agua;
- un sensor del nivel de agua, con contador en el depósito;
- la bomba de alimentación que envía el agua del río.

El código del programa "procedural" consistirá en crear un conjunto de variables que represente los argumentos de cada componente y después escribir el bucle de operación de la gestión central, verificando los valores leídos y actuando en función del resultado de las pruebas. Veremos que, por un lado, están las variables y por otro, las acciones.

La transición hacia el enfoque orientado a objetos

La programación orientada a objetos es un paradigma de programación que considera los diferentes actores de un sistema como objetos activos y relacionados. El enfoque orientado a objetos es mucho más cercano a la realidad.

En nuestro ejemplo, el usuario abre el grifo; este libera presión y el agua fluye desde el depósito hasta el lavabo; el sensor/flotador del depósito llega a un nivel que activa la bomba; el usuario cierra el grifo; alimentado por la bomba, el depósito del agua se rellena y el sensor/flotador alcanza un nivel que detiene la bomba.

En este enfoque, se comprueba que los objetos interactúan; no existe ninguna operación central que defina dinámicamente el funcionamiento de los objetos. En su lugar, hubo un análisis funcional que condujo a la creación de diferentes objetos, su montaje y a establecer sus relaciones.

El código del programa "orientado a objetos" va a seguir esta realidad, proponiendo los objetos descritos anteriormente, pero definiendo los métodos de intercambio adecuados entre estos objetos que conducirán al funcionamiento esperado.

Los conceptos en la orientación a objetos son muy próximos a la realidad...

Las características de la POO

1. El objeto, la clase y la referencia

a. El objeto

El objeto es el elemento básico de la POO. El objeto es la unión:

- de una lista de variables de estado,
- de una lista de comportamientos,
- de una identificación.

Las variables de estado cambian durante el ciclo de vida del objeto. Supongamos el caso de un lector de música digital. Cuando se compra el aparato, los estados de este objeto podrían ser:

- memoria libre = **100%**
- tasa de carga de la batería = **bajo**
- aspecto exterior = **nuevo**

A medida que se usa, sus estados van a cambiar. Rápidamente, la memoria libre va a descender, la tasa de carga de la batería cambiará, así como el aspecto exterior, en función del cuidado que tenga el usuario con el aparato.

Los comportamientos del objeto definen lo que puede hacer este objeto: Reproducir la música - Ir a la pista siguiente - Ir a la pista anterior - Subir el volumen, etc. Una parte de los comportamientos del objeto son accesibles desde el exterior del objeto: botón Play, botón Stop, etc. y otra parte solo desde el interior: lectura de la tarjeta de memoria, decodificación de la música desde el archivo, etc. Hablamos de "encapsulación" para definir un límite entre los comportamientos accesibles desde el exterior y los comportamientos internos.

La identificación de un objeto es información, separada de la lista de estados, que permite diferenciar el objeto de sus congéneres (es decir, de otros objetos del mismo tipo). La identificación puede ser un número de referencia o una cadena de caracteres única, formada durante la creación del objeto; también puede ser una dirección de memoria. En realidad, su forma depende totalmente de la problemática asociada.

El objeto POO puede estar relacionado con una entidad real, en nuestro caso el lector digital, pero también puede estar relacionado con una entidad totalmente virtual, como una cuenta de cliente, una entrada en un directorio telefónico, etc. La finalidad del objeto informático es gestionar la entidad física o emularla.

Incluso si esto no está en su naturaleza básica, el objeto se puede hacer persistente, es decir, que sus estados se pueden registrar en un soporte que memoriza la información de manera

intemporal. A partir de este registro, el objeto se podrá recrear cuando el sistema lo necesite.

b. La clase

La clase es el "molde" a partir del que se va a crear el objeto en memoria. La clase contiene los estados y comportamientos comunes de un mismo tipo. Los valores de estos estados estarán contenidos en sus objetos.

Todas las cuentas corrientes de un mismo banco contienen los mismos argumentos (datos de contacto del titular, saldo, etc.) y todos tienen las mismas funciones (pago a débito o crédito, etc.). Estas definiciones deben estar contenidas en una clase y, cada vez que un cliente abra una nueva cuenta, esta clase servirá de modelo para crear el objeto cuenta.

La pantalla de los reproductores de música digitales ofrece un mismo modelo en diferentes colores, con tamaños de memoria modulables, etc. Cada uno de los dispositivos de visualización es un objeto fabricado a partir de la información de una única clase. Durante su realización, se seleccionan los atributos del aparato en función de criterios estéticos y comerciales.

Una clase puede contener muchos atributos. Pueden ser de tipo primitivo - enteros, caracteres, etc. - y también de tipos más complejos. De hecho, una clase puede contener una o varias clases de otros tipos. En este caso, hablamos de composición o incluso de "fuerte acoplamiento". La destrucción de la clase principal implica, evidentemente, la destrucción de las clases que contiene. Por ejemplo, si una clase hotel contiene una lista de habitaciones, la destrucción del hotel implica la destrucción de las habitaciones.

Una clase puede hacer "referencia" a otra clase; en este caso, el acoplamiento se llama "débil" y los objetos pueden vivir de manera independiente. Por ejemplo, su PC está conectado a su impresora. Si su PC se estropea, su impresora funcionará con el nuevo PC. Hablamos de asociación.

Además de sus atributos, la clase también contiene una serie de "comportamientos", es decir, una serie de métodos con firma y código adjunto. Estos métodos se "copian" directamente en los objetos y se utilizan tal cual.

Se declara la clase y su contenido en un mismo archivo fuente con ayuda de una sintaxis que estudiaremos en detalle. Los desarrolladores C++ aprecian el hecho de que además no exista, por un lado, una parte de definiciones y por otra, una parte de implementaciones. De hecho, en C# el archivo de programa (de extensión .CS) contiene las definiciones de todos los estados y eventualmente un valor "inicial" y las definiciones e implementaciones de todos los comportamientos. Veremos que en C# existe una solución que permite definir una clase en varios archivos, para que las personas que participan en un mismo proyecto puedan trabajar en paralelo, sin que una persona afecte al trabajo de otra.

c. La referencia

Los objetos se construyen a partir de la clase, por un proceso llamado instanciación y, por tanto, cualquier objeto es una instancia de una clase. Cada instancia empieza con una ubicación única en memoria. Los desarrolladores conocen esta ubicación en memoria con el nombre de puntero. C y C++ se han convertido en una referencia para los desarrolladores C# y Java.

Cuando el desarrollador necesita un objeto durante una operación, debe:

- declarar y nombrar una variable del tipo de la clase que va a utilizar;
- instanciar el objeto y registrar su referencia en esta variable.

Cuando se realiza esta instanciación, el programa accederá a las propiedades y métodos del objeto utilizando la variable que contiene su referencia. Cada instancia es única. Por el contrario, varias variables pueden "apuntar" a una misma instancia. Cuando ninguna variable apunta a una instancia dada, el recolector de basura registra esta instancia como instancia para ser destruida.

2. La encapsulación

La encapsulación consiste en crear un tipo de caja negra, que contiene internamente un mecanismo protegido y externamente un conjunto de comandos que van a permitir manipularla. Este juego de comandos se hace de tal manera que será imposible alterar el mecanismo, protegido frente a casos de uso incorrecto. La caja negra será tan opaca que será imposible para el usuario intervenir directamente sobre el mecanismo.

Como habrá entendido, la caja negra no es otra cosa que un objeto con métodos públicos de "alto nivel", que controlan con rigor los argumentos que se pasan antes de utilizarlos en una operación. Respetando el principio de encapsulación, el usuario del objeto jamás podrá acceder "directamente" a sus datos. Gracias a este control, su objeto se utilizará correctamente, por lo que será más fiable y su puesta a punto más sencilla. En el mundo Java, los métodos que permiten acceder en modo lectura a los datos son los descriptores de acceso, y los que permiten acceder en modo escritura son los modificadores. Veremos que C# ofrece una solución muy elegante, las propiedades, que permiten conservar el lado práctico del acceso directo a los datos del objeto, respetando los conceptos principales de la encapsulación.

3. La herencia

Otra noción importante de la POO afecta a la herencia. Para explicarla, imaginemos que debemos construir un sistema de gestión de los diferentes tipos de empleados de una empresa. Como resultado de un análisis rápido, se extrae una lista de propiedades comunes a todos los puestos. De hecho, aunque el empleado sea temporal, directivo, mando intermedio o incluso director, siempre tiene un nombre, un apellido y un número de seguridad social. A esto se le llama generalización; consiste en factorizar los elementos

comunes de un conjunto de clases en una clase más general, llamada superclase en Java y "clase de base" en C# y C++. La clase que hereda de la superclase, se llama subclase o clase heredada.

Por tanto, la herencia va a evitar la redundancia de la información entre los diferentes tipos, creando lo que se llama una herencia de clases. Vamos a crear una clase de base, llamada Empleado, que contiene la información común a todos. Después crearemos una clase llamada Ejecutivo, que heredará de esta clase de base Empleado. Ejecutivo heredará los miembros de Empleado (o más precisamente los miembros que Empleado haya querido publicar) y añadirá a esta lista de miembros los aspectos específicos del tipo ”ejecutivo”. En este caso, se dice que Ejecutivo extiende Empleado o que Ejecutivo hereda de Empleado.

Podemos continuar la herencia diseñando la clase Ingeniero, que hereda de Ejecutivo la cual hereda, por su parte, de Empleado. Desde un objeto Ingeniero tendremos acceso a los datos de sus dos clases de base, y así sucesivamente, hasta la más específica.

En C++ es posible heredar de varias clases al mismo tiempo. Esto puede provocar problemas no resolubles cuando una clase hereda de varias clases las cuales heredan, ellas mismas, de una misma clase de base. Para evitar esto, tanto C# como Java limitan la herencia por nivel a una única clase (por el contrario, puede haber varias herencias en cascada).

Abordaremos más adelante las interfaces, que son un tipo de clases "sin código" que especifican los comportamientos de los objetos, y veremos que una clase C# o Java podrá heredar de tantas interfaces al mismo tiempo como quiera.

La herencia no se limita a la reutilización de atributos. También afecta a los comportamientos. Una clase heredada naturalmente puede añadir nuevos comportamientos respecto a sus aspectos específicos. Pero también puede sustituir los comportamientos de la clase de base por los suyos. Imaginemos una clase de base que representa un animal con un comportamiento "gritar". Este comportamiento no tiene ningún sentido si el animal no se especifica. La clase Perro hereda de la clase Animal y substituye el comportamiento "gritar" de la clase de base por un comportamiento "gritar" adaptado a su tipo: "guau guau". La clase Gato no conserva el mismo método y sustituye este comportamiento "gritar" del animal por un simpático "miau". Si la aplicación gestiona una tabla de tipo Animal que contiene un perro y un gato y llama al método "gritar" de sus dos entradas, verá primero un "guau guau" y después un "miau".

C# se distingue de Java en que ofrece más opciones para el control de la substitución.

4. El polimorfismo

El polimorfismo de los objetos está muy relacionado con la herencia. La raíz etimológica de la palabra conduce a pensar de manera natural que el objeto puede adoptar varias formas. Para entender en qué medida esto es posible, retomemos nuestro objeto Ingeniero. Hereda de la clase Ejecutivo y, por tanto, un objeto Ingeniero es un tipo de Ejecutivo. Gracias al polimorfismo, en cualquier sitio donde se espere un objeto Ejecutivo se podrá utilizar un objeto Ingeniero. Vayamos más lejos y recordemos que el

objeto Ingeniero hereda de Ejecutivo el cual a su vez hereda de Empleado. Por tanto, también se puede decir que Ingeniero es un tipo de Empleado y en cualquier sitio donde se espere un objeto Empleado se podrá utilizar un objeto Ingeniero, como ya habrá imaginado. La consecuencia práctica es que, por ejemplo, se puede construir una tabla de objetos de tipo Empleado que contenga tantas entradas como miembros del personal haya en la empresa. Después se puede instanciar, para cada objeto de esta tabla, objetos de tipo Ingeniero, Temporal, Comercial, etc., ya que heredan de Empleado. No habrá errores de compilación, porque todos los objetos de la tabla serán de tipos compatibles con Empleado. Se podrán realizar operaciones en esta colección, como la edición de los recibos de nómina, los cálculos de pensiones, etc. Estas operaciones se convierten en operaciones totalmente genéricas, porque consideran cada entrada como un Empleado. En función del objeto que se instancie, la operación básica se sustituye por una operación específica.

Veremos cómo C# ofrece medios muy sencillos y prácticos que permiten saber con certeza si un objeto de un determinado tipo también forma parte de otra familia (operador is) y se puede considerar como tal durante una operación (operador as). También veremos cómo la herencia de interfaces y el polimorfismo pueden estar estrechamente relacionados para que se puedan comunicar los objetos diseñados, aunque hayan sido diseñados con años de diferencia.

El polimorfismo también puede afectar a los comportamientos de un objeto. De hecho, un comportamiento con el mismo nombre puede aparecer varias veces en un objeto, con la condición de que las firmas (los argumentos esperados) sean diferentes y permitan diferenciarlos. Es el compilador el que deduce, de manera estática, en función del código del programa que llama, el comportamiento al que se debe llamar.

5. La abstracción

El comportamiento correcto en POO consiste en establecer relaciones entre objetos lo más abstractas posible. Por ejemplo, una fuente de datos puede provenir de una entrada por teclado, del contenido de un archivo, de una conexión de red, etc. Si determinados métodos de su clase reciben como argumento una sucesión de datos, será interesante abstraerse de su origen, considerándola como un flujo "genérico" (stream).

Otro ejemplo: .NET ofrece diferentes clases para gestionar las colecciones de datos. La elección de la clase mejor adaptada se hará según diferentes criterios, como la rapidez de acceso a una ubicación de la colección o la rapidez de inserción de elementos en cualquier lugar de la colección. Estas optimizaciones se hacen en el código de implementación de cada clase y ofrecen un medio de explotación genérico, común a todos los tipos que abstraen de estas capas básicas. El código del programa permanece prácticamente idéntico, independientemente del tipo de colección que se utilice.

La programación orientada a objetos y su implementación en C# ofrecen diferentes medios como las interfaces, las clases abstractas, los flujos y los iteradores para realizar la abstracción de los objetos lo mejor posible.

El desarrollo orientado a objetos

1. Especificaciones del software

La primera etapa de un desarrollo de software consiste en generar su descripción, utilizando un lenguaje comprensible por el cliente que lo ha solicitado y por los desarrolladores, en la que se expliquen las funcionalidades del producto a realizar. Este documento es muy importante, porque va a definir los límites del programa, formalizando las necesidades, requisitos y restricciones. Definirá los diferentes tipos de usuarios y sus posibles interacciones en función de sus permisos, etc. Durante la redacción de este documento es necesario impregnarse de la cultura de la empresa y hablar con todas las personas que intervengan. Las especificaciones son el elemento básico de la modelización. Gracias a su particular sentido, como consecuencia de su realización, se podrá evitar cualquier tipo de controversia posterior.

2. Modelización y representación UML

La modelización es la fase esencial del desarrollo de una aplicación. Se basa en las especificaciones y consiste en analizar y descomponer un proceso en varios elementos sencillos. Permite "diseñar los planes" de los componentes a realizar, verificar que serán evolutivos, robustos, fiables y que su funcionamiento conjunto realizarán el objetivo solicitado. La modelización oculta los detalles, para presentar lo principal. Hablamos de **abstracción**.

Se plantea la cuestión de representar de manera "normalizada" la modelización, cuestión a la que UML responde de manera satisfactoria.

Dediquemos un poco de tiempo a ver cómo nació este lenguaje de modelización de objetos que es UML.

La explosión de la programación orientada a objetos y el crecimiento de la complejidad de los programas generaron una multiplicación de los métodos orientados a objetos a principio de los años 90. Entre estos métodos, podemos mencionar:

- Booch’91 de Grady Booch.
- Object Modeling Technique (OMT) de James Rumbaugh en 1991.
- Object-Oriented Software Engineering (OOSE) de Ivar Jacobson en 1992.

La representación de manera estándar del funcionamiento y arquitectura de un sistema, así como la comunicación de sus objetos, se convirtió rápidamente en algo necesario para:

- Estructurar de manera evolutiva sus componentes.
- Aumentar su fiabilidad y la seguridad de su conjunto.
- Facilitar su mantenimiento.
- Transmitir y garantizar su comprensión por parte de otros equipos.
- Reutilizar sus componentes.

En 1997, con un objetivo de unificación, los diferentes métodos empezaron a centrarse en poner en común sus puntos fuertes, validados por el feedback de los usuarios, para alumbrar un modo de representación llamado Unified Modeling Language (UML), en su versión 1.0.

UML es un lenguaje de modelización de objetos no propietario. Se rige por el Object Management Group (OMG) y la norma está disponible gratuitamente en www.uml.org. UML es un lenguaje gráfico, mientras que C++, Java y C# son lenguajes textuales.

Los diseñadores de software utilizan UML para representar sus modelos en forma de gráficos, también llamados vistas. Estas vistas contienen diagramas de diferentes tipos que explican, bajo ángulos diferentes, el contenido y el funcionamiento de la aplicación. Algunas veces es necesario tener varias vistas para representar y entender totalmente un modelo.

A continuación, se enumeran los nueve principales tipos de diagramas UML y sus objetivos:

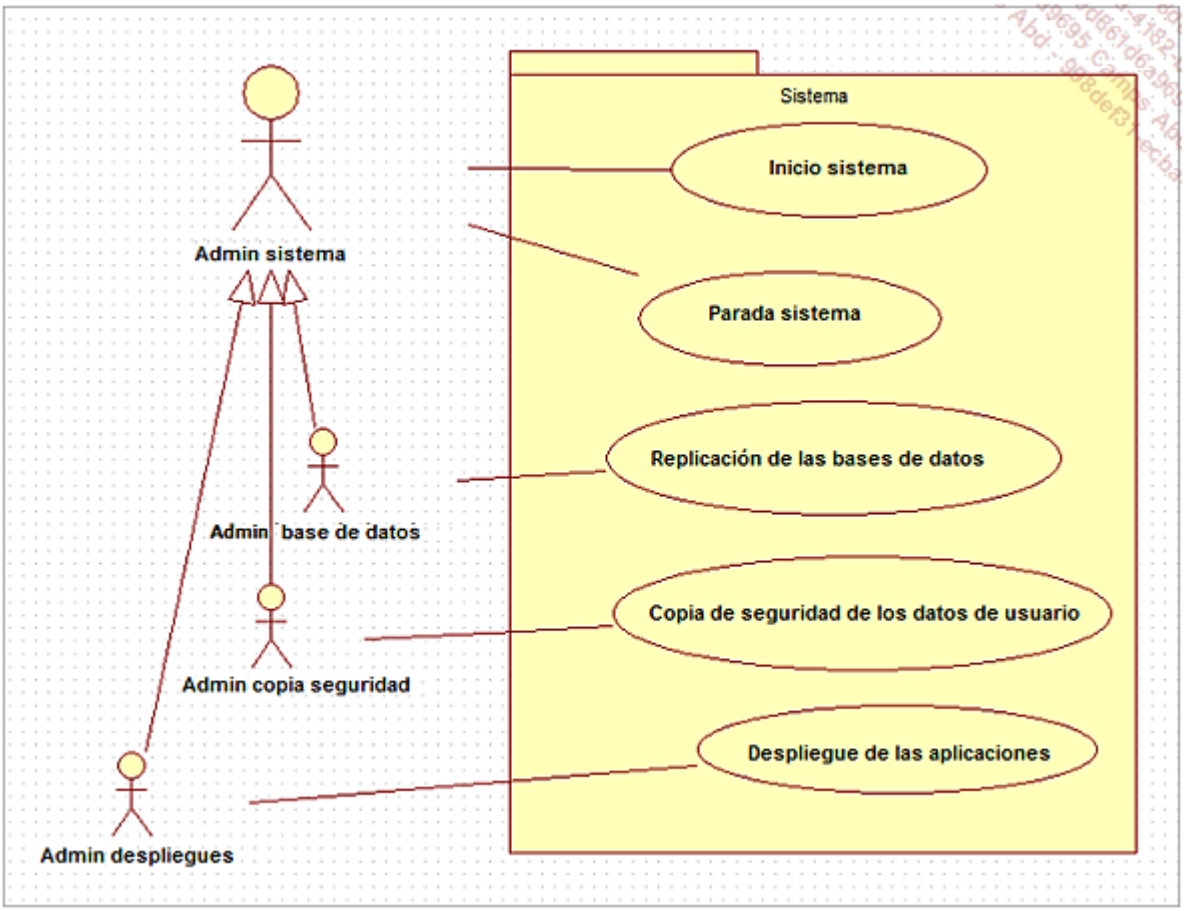
- Diagramas de casos de uso. Describen los servicios ofrecidos por el sistema, desde el punto de vista del usuario. Estas vistas ponen en escena a los actores, que pueden ser humanos o representar a otros sistemas.
- Diagramas de objetos. Muestran el estado de una aplicación en un instante dado, nombrando a las instancias de las clases.
- Diagramas de secuencia. Muestran las interacciones entre los objetos durante la ejecución del programa. El acento se pone en el orden de estas interacciones dentro de esta representación temporal.
- Diagramas de clases. Capturan la estructura estática de organización de las clases.
- Diagramas de componentes. Son vistas modulares de la aplicación que agrupan las clases que colaboran.
- Diagramas de despliegue. Modelizan el aspecto hardware de la aplicación.
- Diagramas de colaboración. Muestran cómo se organizan los objetos para trabajar en conjunto. El acento se pone en las comunicaciones existentes entre los objetos.
- Diagramas de estados-transiciones. Representan el comportamiento de un objeto en forma de un autómata de estados finales.
- Diagramas de actividad. Representan el flujo de ejecución de un proceso u operación.

La mayor parte de los desarrolladores solo utilizan un subconjunto de diagramas de UML, principalmente los diagramas de casos de uso, clases y secuencias.

Hay muchas aplicaciones de software libre que permiten construir diagramas UML, como **Modelio**, que se puede descargar desde la dirección: http://archive.modeliosoft.com/es/products-es/modelio-free-edition-es.html

a. Diagramas de casos de uso

El papel de los diagramas de casos de uso es delimitar el perímetro de la aplicación, indicando sus "actores" y las diferentes posibilidades que puede haber en el sistema. Un caso de uso representa un servicio funcional de la aplicación descrito en las especificaciones. El caso de uso se acompaña de un texto que lo describe de manera precisa, con sus condiciones de inicio, su funcionamiento normal y el resultado de su ejecución. Para precisar el caso de uso, se pueden añadir diagramas de secuencias o actividades.



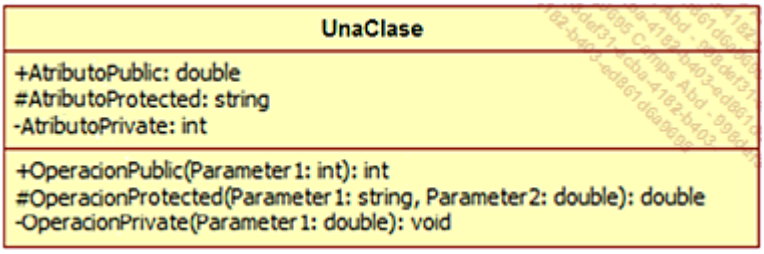
Esta ilustración representa cuatro actores que tienen permisos para efectuar determinadas operaciones en un sistema informático. Un actor un poco especial, Admin systema, hereda de los otros tres y, por tanto, tiene permisos para realizar sus operaciones.

b. Diagramas de clases

La agrupación de objetos del mismo tipo permite factorizar sus atributos y sus comportamientos. La representación gráfica se realiza en un diagrama de clases. En este diagrama se definen, sin que se precise el número de instancias, los componentes finales de la aplicación. Las relaciones entre las clases también se representan para cada tipo de relación con un signo gráfico diferente. El conocimiento de las diferentes notaciones utilizadas es primordial para una transcripción correcta utilizando código C#. Los diagramas de clases no presentan los aspectos dinámicos y temporales.

Una clase se representa por un rectángulo, dividido verticalmente en tres partes:

- En la parte superior: el nombre de la clase
- En el medio: los atributos de la clase (las variables)
- En la parte inferior: los comportamientos de la clase (los métodos)



Los miembros de la clase (atributos y comportamientos) están precedidos por un signo (+, #, -) que indica su accesibilidad. Esta información permite gestionar la encapsulación y herencia, descritas anteriormente en este capítulo.

El signo + indica que este miembro de la clase es accesible por todos, sin restricción.

El signo # indica que este miembro de la clase es accesible solo por sus clases heredadas.

El signo - indica que este miembro de la clase es privado y, por tanto, solo será utilizado por la propia clase para su funcionamiento interno. Observe que un objeto puede acceder a los miembros privados de otro objeto, si ambos son del mismo tipo.

Después del signo de accesibilidad y el nombre del atributo, aparece un ':', seguido del tipo del atributo. Este tipo puede ser "clásico" (entero, carácter, decimal, etc.) o más complejo como, por ejemplo, otra clase. Eventualmente, el signo '=' seguido de un valor puede terminar la definición. En este caso, se tratará del valor signado al atributo durante su instanciación. Veremos que, sin esta declaración, se inicializa un valor por defecto durante la instanciación del objeto. Por ejemplo, un atributo de tipo entero se inicializará automáticamente a 0.

Si respeta la regla de la encapsulación, no debería aparecer ningún atributo precedido de un +.

De la misma manera que los atributos, los métodos de la clase están precedidos por su tipo de acceso. Después del nombre del método, aparece la lista entre paréntesis de los argumentos nombrados y tipados. Puede que no haya argumentos. Para terminar, el tipo de retorno aparece precedido por un ':'. Si no se devuelve ningún tipo, se define el tipo void.

Atención, esta sintaxis UML que define un comportamiento difiere de la de C# en que define el método asociado.

- En UML: +Suma(a:Int, b:Int):Int
- En C#: public int Suma(int a, int b)

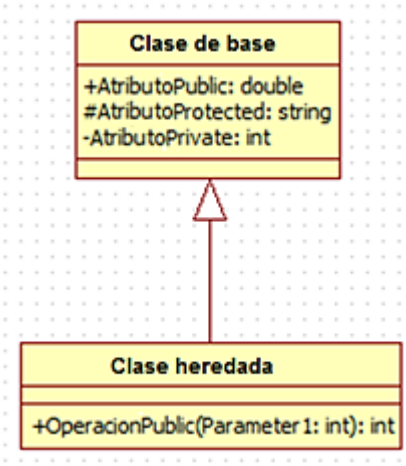
Recordemos que varios métodos pueden tener el mismo nombre. Esta forma de polimorfismo solo es posible si difieren los argumentos de los métodos del mismo nombre.

Las relaciones entre las clases

Las clases pueden estar relacionadas de manera más o menos fuerte y los diagramas representan gráficamente estas diferencias con sombreado.

Representación de una herencia

Esta especialización se representa en UML por una flecha, con el triángulo cerrado, que va de la clase derivada (subclase, clase heredada o incluso clase especializada) hasta la clase madre (clase de base o superclase).

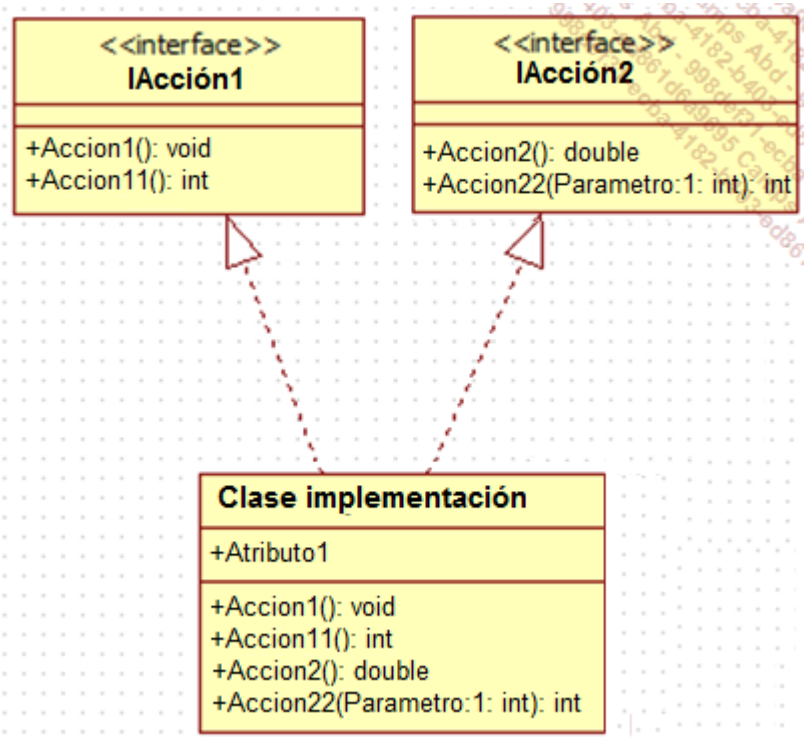


La clase heredada es la extensión de la clase de base. El inicio de la porción de memoria asignada a una clase heredada apunta a la clase de base; los aspectos específicos de la clase heredada vienen a continuación. Gracias a esta organización de memoria, se puede decir que como una clase heredada es de un tipo de clase de base, se puede utilizar en cualquier sitio donde se necesite su clase de base.

La clase heredada se beneficia de los miembros públicos (+) y protegidos (#) de la clase de base.

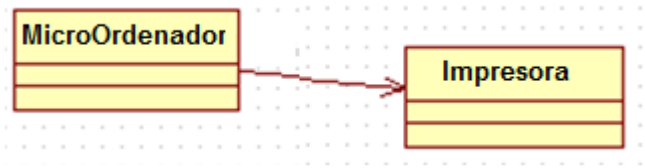
Representación de una realización

Una clase puede implementar varias interfaces. Como veremos más adelante, esto es un compromiso para tener en cuenta una lista de métodos determinada. UML representa esta "realización" con una línea punteada, terminada por una flecha no punteada y en forma de triángulo dirigido hacia la interfaz.



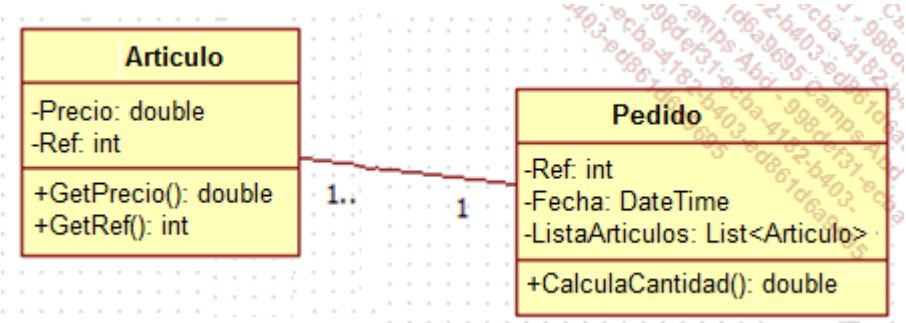
Representación de una relación sencilla (asociación)

Una clase puede contener una referencia a otra y de esta manera acceder a sus servicios. La representación de esta asociación de navegación es una flecha abierta que apunta a la clase referenciada. Las líneas del ciclo de vida de las dos clases son independientes.



Representación de una relación bidireccional

Se pueden asociar dos tipos de clases durante una determinada operación. La asociación se representa por una sencilla línea completa. Las dos clases no tienen ninguna relación "parental"; sus líneas de ciclo de vida son totalmente independientes. Por ejemplo, una clase Pedido contiene una lista de Artículos. No es el Pedido quien crea los Artículos; si el pedido se elimina, los Artículos permanecen.

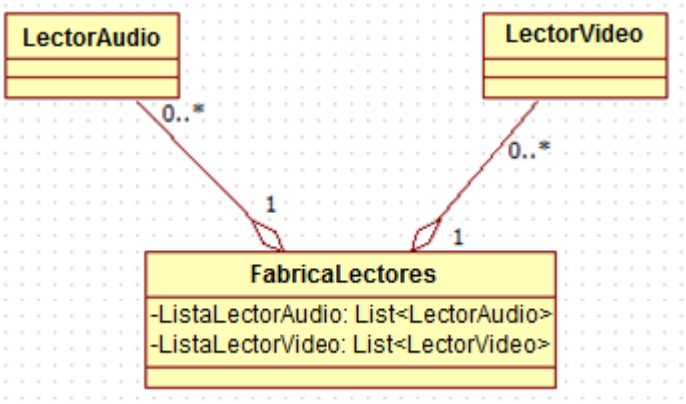


Una restricción enumerativa puede indicar los términos de la asociación. En el esquema anterior, se debe leer que un pedido contiene al menos un artículo y puede contener un número indefinido de artículos (1..*). En el otro sentido, un artículo está en un pedido. Hablamos de índice de cardinalidad o de multiplicidad.

Representación de una relación de tipo agregación

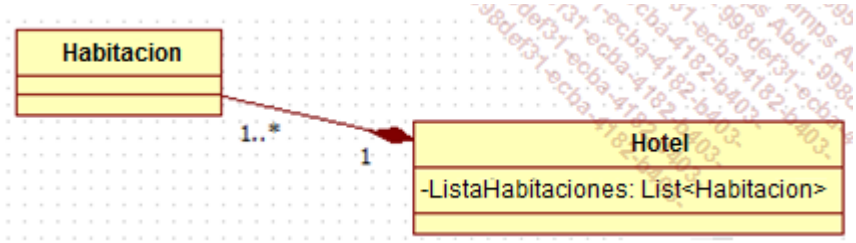
Una línea con un rombo relaciona las dos clases. El rombo está junto en la clase "contenedor". Si el rombo está vacío quiere decir que los objetos creados por esta clase le sobrevivirán.

Por ejemplo, un fabricante de lectores digitales tendrá este tipo de relación con sus productos fabricados. Incluso si echa el cierre, los reproductores seguirán funcionando. Hablamos de agregación por referencia, porque hay varios objetos independientes en memoria y determinados objetos memorizan las referencias hacia los otros.



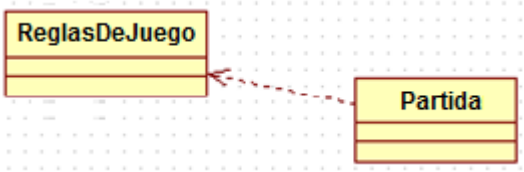
Representación de una relación de tipo composición

Si el rombo está relleno, entonces la relación es fuerte. Retomemos el ejemplo del hotel. Si el hotel se destruye, sus habitaciones también. Hablamos de agregación compuesta o de agregación por valor. La duración del ciclo de vida de los componentes y de los compuestos es idéntica.



Las dependencias entre clases

Partida de un juego de mesa depende de las reglas de este juego. Aquí no hay relación de asociación o composición; sencillamente una dependencia entre la partida y las reglas escritas. UML representa esta relación con una línea punteada.



c. Enumeraciones

Con mucha frecuencia, en programación queremos limitar los posibles valores de una variable y restringirlos a un conjunto determinado de valores. Por ejemplo, puede necesitar un tipo de datos que almacene un día de la semana y nada más. No sucede nada si el compilador muestra un error cuando intenta insertar una información diferente utilizando líneas de programación, por ejemplo, el nombre de un mes. Es posible configurar esta seguridad gracias a las enumeraciones, que permiten definir una lista de valores posibles. No importa cómo codifique estos valores el compilador, lo que cuenta es que prohíbe cualquier otro valor.

En UML, una enumeración se representa como una clase, cuyo nombre se precede de <<enumeration>>. Después, se utiliza directamente para tipar los atributos de las clases que lo necesitan.

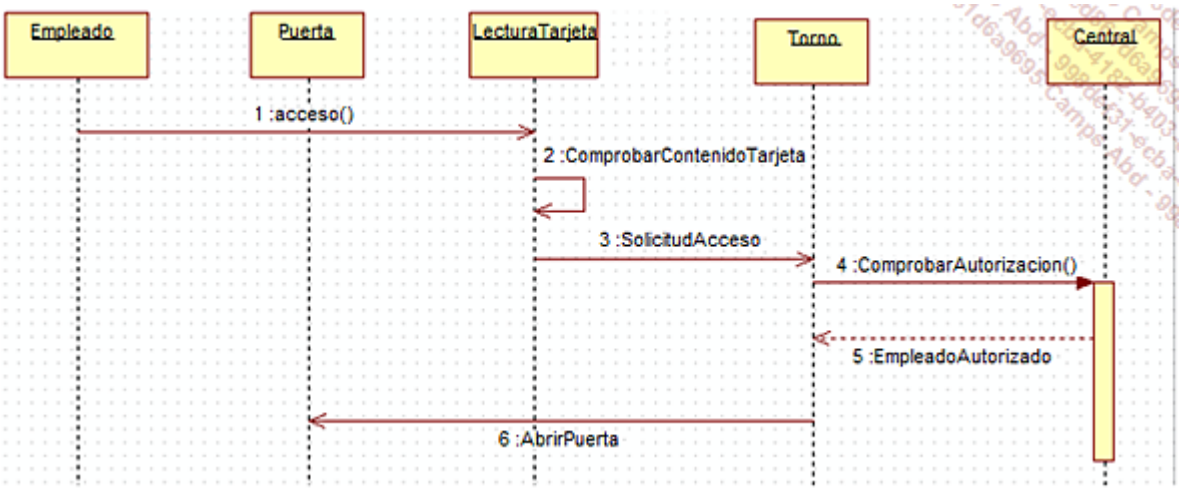
Este diagrama representa una clase Abonado que contiene varios atributos, entre los cuales hay un tipo enumeración DíaDeLaSemana, que enumera una serie de valores para cada día de la semana. Generalmente, los compiladores asignan un juego de valores que empieza por 0 y va aumentando de uno en uno; pero no importa, porque el código nunca hará referencia a estos valores de sustitución (lo que provocaría un error de compilación). El código utilizará directamente los elementos de la lista...

```
if( miCliente.DiaFormacion == DiaDeLaSemana.Lunes)
{
//...
}
```

d. Diagramas de secuencias

El diagrama de secuencia es una representación cronológica de las interacciones entre los objetos para realizar un caso de uso. Describe la lista de objetos en juego, sus líneas de ciclo de vida y la cronología de las interacciones.

A continuación se muestra un ejemplo de un diagrama de secuencia. Representa el encadenamiento de las interacciones que permiten abrir una puerta después de reconocer la tarjeta de un usuario.



Cada objeto que interviene en la interacción se representa con un rectángulo que contiene el nombre del actor o el nombre de una instancia particular seguida del nombre de la clase asociada. Bajo este rectángulo se añade una línea vertical discontinua que representa la línea del ciclo de vida del objeto. Entre estas líneas del ciclo de vida, se añaden flechas horizontales que representan las interacciones. Cada flecha está numerada y nombrada. Este nombre generalmente se corresponde con el de un método implementado en el objeto receptor del mensaje. La secuencia siempre empieza por la parte superior y normalmente por la izquierda.

La interacción entre objetos

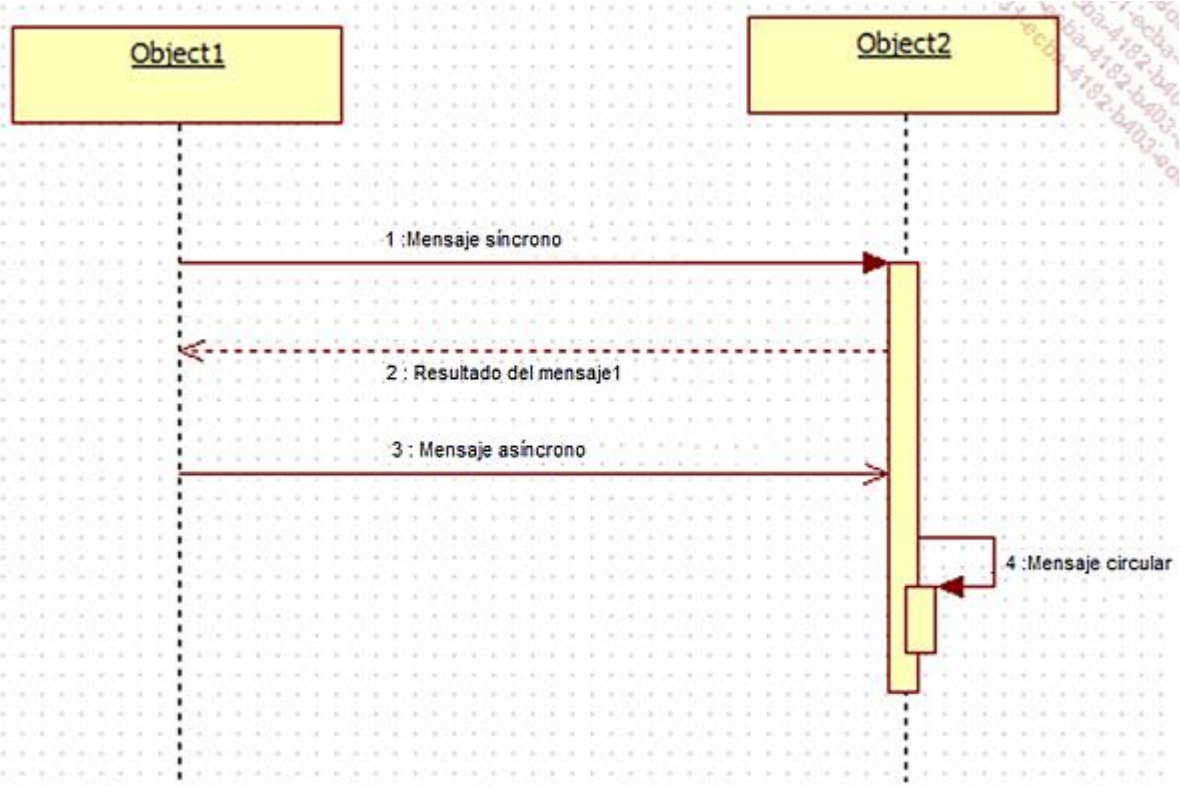
En el lenguaje UML, la interacción entre objetos se realiza mediante mensajes. En particular, la mayor parte del tiempo estos mensajes son sencillas llamadas a métodos de las instancias de clase. Es el comportamiento del emisor en relación con el retorno el que va a especificar el tipo de mensaje y a influir completamente en la codificación que se debe implementar.

De hecho, hablamos de mensaje:

- Síncrono, cuando el emisor espera la respuesta del receptor y bloquea la ejecución del programa.
- Asíncrono, cuando la respuesta del receptor llega más tarde al emisor.

Un objeto también se puede enviar mensajes; hablamos de interacción interna o de mensaje circular.

La lectura del diagrama de secuencia permite conocer la naturaleza del mensaje intercambiado en un momento dado. La representación gráfica de los mensajes en el diagrama de secuencia es la siguiente.



Observe la forma de la flecha que termina el mensaje:

- flecha cerrada, para un mensaje síncrono;
- flecha abierta, para un mensaje asíncrono.

3. Redacción del código y pruebas unitarias

La redacción del código se debe llevar a cabo respetando los conceptos principales de la programación orientada a objetos.

En programación, algunos problemas aparecen de manera recurrente. Antes de intentar inventar sus propias soluciones, puede ser buena idea ver lo que otros desarrolladores han hecho para responder a problemas parecidos. Los patrones de diseño (design patterns) describen soluciones sencillas y elegantes en programación orientada a objetos para responder a estos problemas. Estos design patterns no son librerías de código, sino métodos para resolver el problema; la implementación en un lenguaje determinado es responsabilidad del desarrollador. Más adelante veremos las implementaciones en C# de dos patrones de diseño. Para ir más allá, puede hacerse con el catálogo de los design patterns "clásicos", llamado catálogo GoF por las iniciales de Gang of Four, término que hace referencia a los cuatro co-autores: Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides.

También es muy aconsejable realizar un control de calidad a medida que se construyen las clases. Los entornos de desarrollo actuales normalmente ofrecen un framework en el que el desarrollador escribe un juego de pruebas unitarias para sus objetos, que permiten verificar el funcionamiento y garantizar que no haya regresión después de las modificaciones u operaciones de mantenimiento. Estas pruebas se podrán automatizar, lanzándolas después de la compilación, y los resultados se formatearán por la propia herramienta de desarrollo. Esto se tratará en el capítulo Las pruebas.

Después de estas pruebas unitarias, vienen las pruebas de integración, que permiten comprobar que los módulos se comunican correctamente entre ellos. Estas pruebas también se pueden automatizar en el entorno de desarrollo.

Ahora la aplicación se puede probar en su conjunto. Pero como este conjunto está formado por módulos ya verificados y estabilizados por las pruebas unitarias y las pruebas de integración, los funcionamientos incorrectos residuales deberían consistir en pequeños ajustes más que en una corrección de errores importantes.

Esta última fase de puesta a punto es previa a la puesta a prueba del producto. Esta comprobación se base en una serie de pruebas, en los escenarios escritos y validados por el cliente y el desarrollador, para controlar que el software es conforme a las especificaciones.

Introducción

El framework .NET es un entorno de ejecución y desarrollo principalmente diseñado para entornos Windows.

Entorno de ejecución

Las aplicaciones diseñadas para funcionar con el framework .NET prácticamente nunca se comunican directamente con el sistema operativo. Se dice que su código es de tipo gestionado (managed), porque es el framework .NET el que va a gestionar su ejecución dentro de su Common Language Runtime (CLR), cuyo núcleo se encuentra en la DLL mscoree.dll.

Incluso si los archivos generados por los compiladores para .NET tienen extensiones .EXE y DLL, el código contenido en estos archivos no es lenguaje directamente ejecutable por el microprocesador. Es un código intermedio llamado Microsoft Intermediate Language (MSIL), totalmente independiente del lenguaje de desarrollo original. Cuando se hace doble clic en el icono de una aplicación .NET, la ejecución lee el encabezado del archivo, que le informa del modo de funcionamiento gestionado, y adapta el tipo de ejecución en consecuencia. Durante la ejecución, el código MSIL se compila "sobre la marcha" en código máquina, por medio de un Just In Time Compiler (o JIT Compiler).

Esta operación, adicional respecto a la ejecución de una aplicación "unmanaged", es decir, que contiene directamente el código máquina, la mayor parte del tiempo es imperceptible a efectos de rendimiento en los ordenadores actuales. Además, la imagen realizada se guarda en una zona memoria especial, llamada "caché". Esta copia se reutiliza directamente por otros procesos de carga de la aplicación.

La elección de los lenguajes

La codificación de las aplicaciones compatibles con el framework .NET no implica un lenguaje particular. Microsoft ha publicado las especificaciones de la CLI (Common Language Infraestructura), que permite crear compiladores compatibles con la CLR (Common Language Runtime). De esta manera, incluso si C# es el lenguaje preferido de .NET, es posible construir aplicaciones en Visual Basic .NET, C++ gestionado u otro F#...

Observe que el código MSIL resultado de la compilación puede desensamblarse fácilmente en el lenguaje inicial o en uno de los lenguajes que gestiona .NET. Es posible hacer el resultado de esta acción menos legible realizando una operación de ofuscación.

Utilizar varios lenguajes

Otro punto fuerte de esta arquitectura es que es posible tener en la misma aplicación una parte escrita en C#, otra en Visual Basic, etc. Esta proeza es posible gracias a la CLS (Common Language Specification), que unifica la gestión de los tipos de datos básicos para todos los lenguajes.

Una librería muy completa

El framework .NET ofrece una colección muy amplia de clases, en las que se basan las aplicaciones. Estas clases simplifican considerablemente la gestión de objetos habituales (cadenas de caracteres, valores decimales, etc.), así como la gestión de archivos, interfaces gráficas clásicas (WinForms) o "modernas" (WPF), APIs web, acceso a las bases de datos (ADO.NET), comunicaciones de red (WCF), seguridad, diagnóstico, etc. La lista de clases es muy amplia y además se disponen de manera jerárquica, por lo que el problema normalmente es saber localizarlas.

La mayor parte de las veces, estas clases son extensibles. Esto quiere decir que es posible heredar para aprovechar su comportamiento básico y añadir los aspectos específicos de negocio.

Las librerías de .NET son independientes del lenguaje, lo que permite compartir los mismos tipos, independientemente del origen de los componentes.

Para organizar estas clases, la plataforma utiliza el concepto de espacio de nombres (namespace), que agrupa las clases por objetivos (servicios de la misma naturaleza). Por ejemplo, el namespace System.XML contiene la caja de herramientas para gestionar los datos XML.

Un ensamblado (assembly) es un archivo que contiene uno o varios espacios de nombres, que contienen ellos mismos uno o varios tipos. El ensamblado de C# corresponde al paquete de UML. El desacoplamiento tiene por objeto reducir las dependencias entre los ensamblados. Para que un programa pueda utilizar un ensamblado, este último se debe referenciar en el proyecto y se debe declarar su uso en la parte superior del archivo de código fuente correspondiente con la directiva using, por ejemplo: using System.XML;.

Instalaciones simplificadas

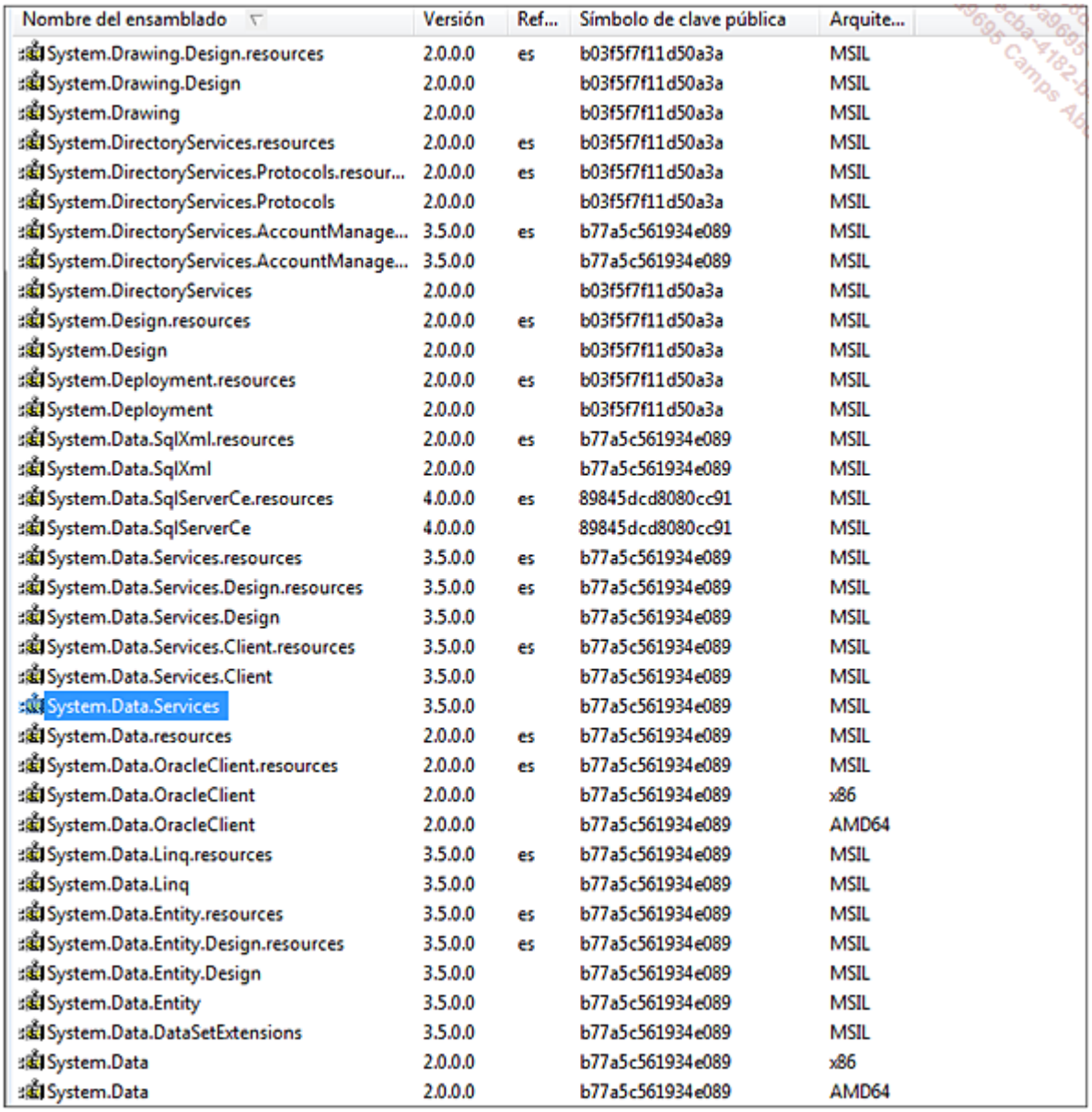
Por defecto, los ensamblados de un proyecto son privados y se instalan en el mismo directorio que el de la aplicación. Por el contrario, si un ensamblado de tipo biblioteca se debe compartir entre varias aplicaciones, entonces se ubicará en un lugar especial llamado Global Assembly Cache (GAC).

Los archivos DLL de ensamblados ubicados en la GAC deben poder mostrar su identidad e integridad, teniendo "nombres fuertes" (strong names), formados por:

- el nombre del ensamblado,
- su versión,
- su "cultura",
- una clave de cifrado pública,
- una firma digital derivada del contenido mismo del assembly.

El nombre compuesto de esta manera puede hacer referencia al ensamblado de manera unívoca y se puede utilizar para comprobar que el contenido inicial no se ha modificado desde su construcción.

La GAC es un directorio de archivos, pero los ensamblados no se pueden ubicar directamente con el explorador de archivos. Hay que utilizar la herramienta **gacutil.exe**.



Nombre del ensamblado	Versión	Ref...	Símbolo de clave pública	Arquite...
System.Drawing.Design.resources	2.0.0.0	es	b03f5f7f11d50a3a	MSIL
System.Drawing.Design	2.0.0.0		b03f5f7f11d50a3a	MSIL
System.Drawing	2.0.0.0		b03f5f7f11d50a3a	MSIL
System.DirectoryServices.resources	2.0.0.0	es	b03f5f7f11d50a3a	MSIL
System.DirectoryServices.Protocols.resour...	2.0.0.0	es	b03f5f7f11d50a3a	MSIL
System.DirectoryServices.Protocols	2.0.0.0		b03f5f7f11d50a3a	MSIL
System.DirectoryServices.AccountManage...	3.5.0.0	es	b77a5c561934e089	MSIL
System.DirectoryServices.AccountManage...	3.5.0.0		b77a5c561934e089	MSIL
System.DirectoryServices	2.0.0.0		b03f5f7f11d50a3a	MSIL
System.Design.resources	2.0.0.0	es	b03f5f7f11d50a3a	MSIL
System.Design	2.0.0.0		b03f5f7f11d50a3a	MSIL
System.Deployment.resources	2.0.0.0	es	b03f5f7f11d50a3a	MSIL
System.Deployment	2.0.0.0		b03f5f7f11d50a3a	MSIL
System.Data.SqlXml.resources	2.0.0.0	es	b77a5c561934e089	MSIL
System.Data.SqlXml	2.0.0.0		b77a5c561934e089	MSIL
System.Data.SqlServerCe.resources	4.0.0.0	es	89845dcd8080cc91	MSIL
System.Data.SqlServerCe	4.0.0.0		89845dcd8080cc91	MSIL
System.Data.Services.resources	3.5.0.0	es	b77a5c561934e089	MSIL
System.Data.Services.Design.resources	3.5.0.0	es	b77a5c561934e089	MSIL
System.Data.Services.Design	3.5.0.0		b77a5c561934e089	MSIL
System.Data.Services.Client.resources	3.5.0.0	es	b77a5c561934e089	MSIL
System.Data.Services.Client	3.5.0.0		b77a5c561934e089	MSIL
System.Data.Services	3.5.0.0		b77a5c561934e089	MSIL
System.Data.resources	2.0.0.0	es	b77a5c561934e089	MSIL
System.Data.OracleClient.resources	2.0.0.0	es	b77a5c561934e089	MSIL
System.Data.OracleClient	2.0.0.0		b77a5c561934e089	x86
System.Data.OracleClient	2.0.0.0		b77a5c561934e089	AMD64
System.Data.Linq.resources	3.5.0.0	es	b77a5c561934e089	MSIL
System.Data.Linq	3.5.0.0		b77a5c561934e089	MSIL
System.Data.Entity.resources	3.5.0.0	es	b77a5c561934e089	MSIL
System.Data.Entity.Design.resources	3.5.0.0	es	b77a5c561934e089	MSIL
System.Data.Entity.Design	3.5.0.0		b77a5c561934e089	MSIL
System.Data.Entity	3.5.0.0		b77a5c561934e089	MSIL
System.Data.DataSetExtensions	3.5.0.0		b77a5c561934e089	MSIL
System.Data	2.0.0.0		b77a5c561934e089	x86
System.Data	2.0.0.0		b77a5c561934e089	AMD64

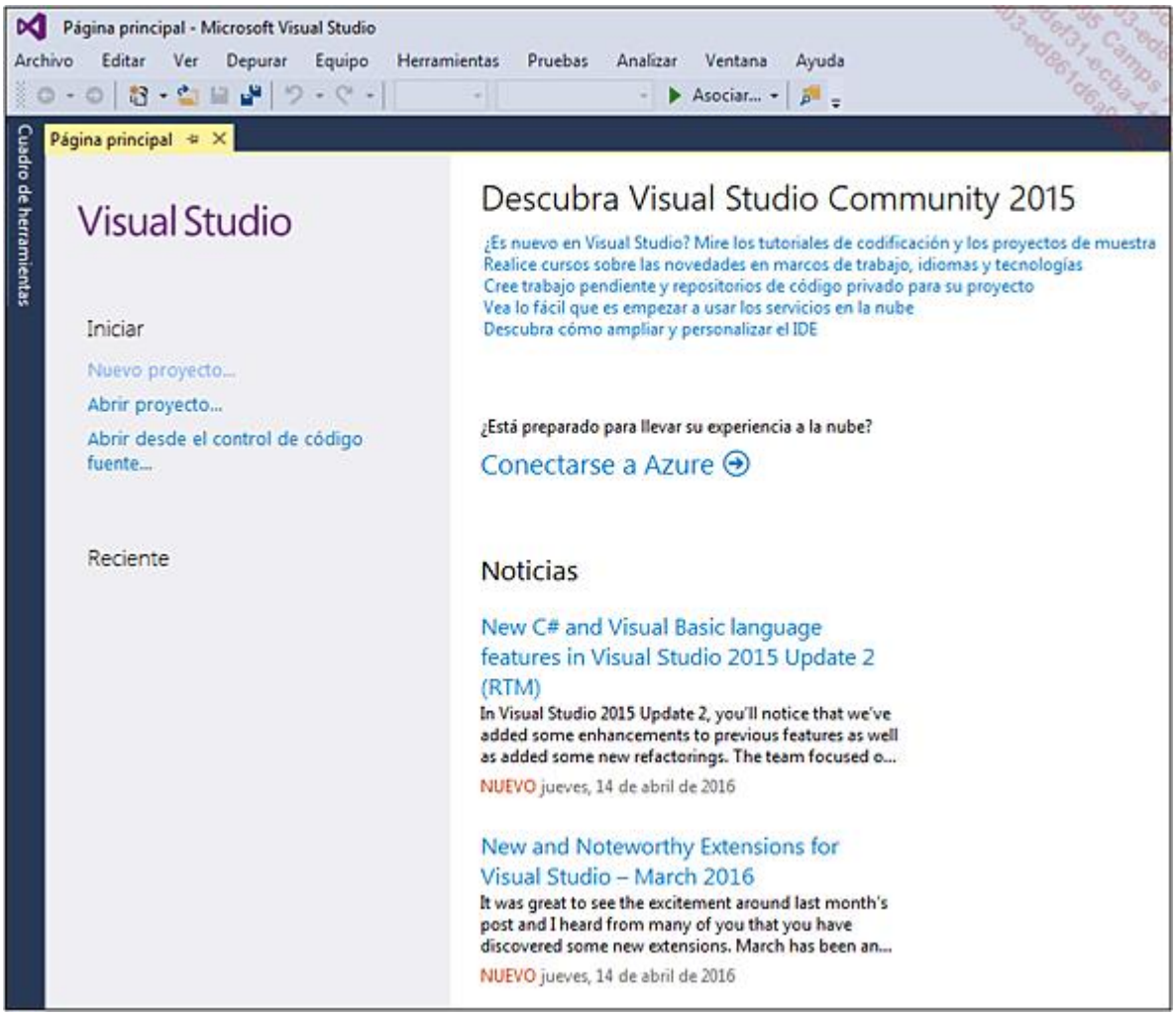
Los componentes de la GAC se referencian principalmente por su nombre y su número de versión. Una aplicación declara el uso de un ensamblado utilizando estos dos datos. Como la GAC puede contener ensamblados con el mismo nombre pero con números de versión diferentes es posible actualizar una DLL sin impactar a la existente. "Bye bye DLL hell!"

Ya no se utiliza la base de registro para describir los módulos, lo que simplifica considerablemente la instalación de las aplicaciones.

Una herramienta de desarrollo completa

El framework .NET ofrece un compilador C# que se puede utilizar directamente por línea de comandos después de haber introducido el código del programa en un editor de texto, como NotePad. El procedimiento es posible pero no muy productivo. Evidentemente, el desarrollador busca utilizar un entorno de desarrollo totalmente integrado, que le ayude durante la escritura del código, le permita realizar la interfaz gráfica, la puesta a punto y el despliegue de su aplicación. Este programa es un IDE (Integrated Development Environment). Integra al menos un editor de código fuente, herramientas que automatizan las compilaciones y un unlocker.

Visual Studio es la herramienta de desarrollo propuesta por Microsoft. Potente, estable y con un entorno amigable, Visual Studio se presenta en varias versiones: Community, Professional y Entreprise. Los ejemplos de este libro se pueden probar con la versión **Visual Studio Community 2015**, que se puede descargar de manera gratuita en https://www.visualstudio.com/es-es/products/vs-2015-product-editions.aspx. Naturalmente, también se podrán probar con las versiones más potentes de Visual Studio.



Descarga e instalación de Visual Studio Community

Con su navegador de Internet, vaya a la siguiente dirección: https://www.visualstudio.com/es-es/products/vs-2015-product-editions.aspx

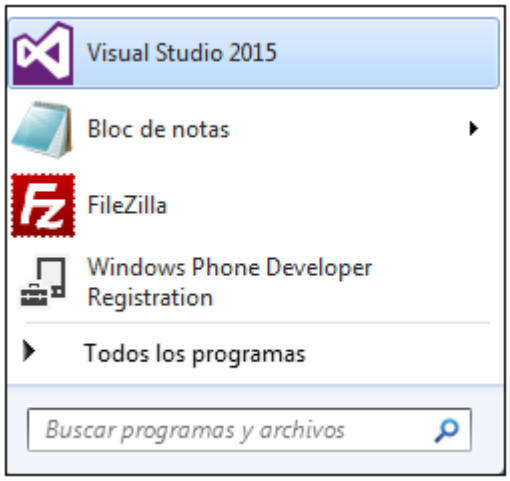
Abra el menú **Descarga** y seleccione **Visual Studio Community**.

Se descarga el archivo vs_community.exe y el navegador le pide que confirme su ejecución.

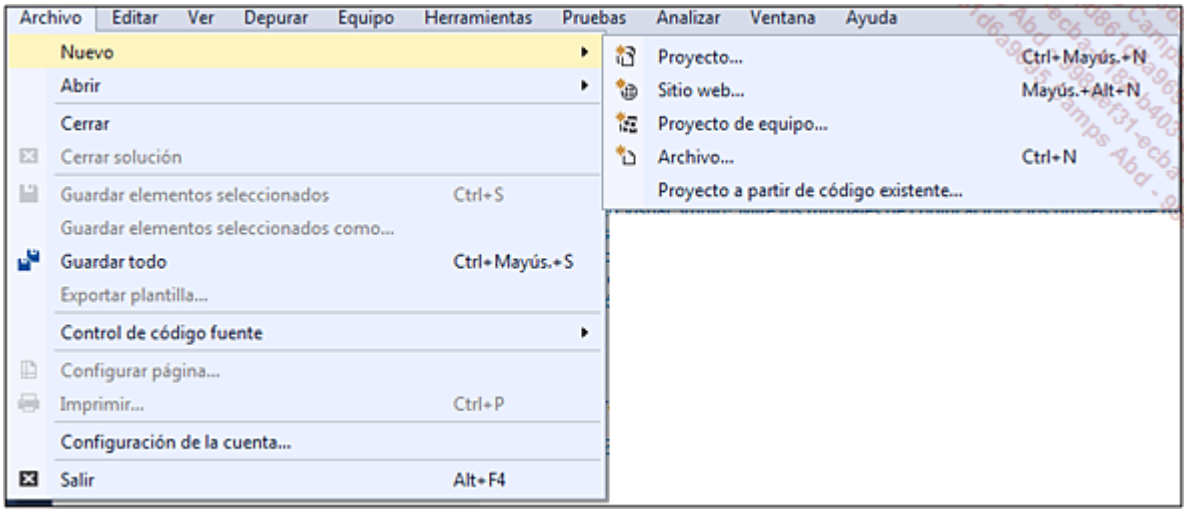
El inevitable Hello World

Incluso si el aprendizaje de Visual Studio se hace al mismo tiempo que se aprende la programación orientada a objetos, no podemos resistirnos más a escribir nuestro primer programa, a saber, el clásico Hello World!

Ejecute Visual Studio desde el menú **Iniciar - Todos los programas - Visual Studio 2015**.



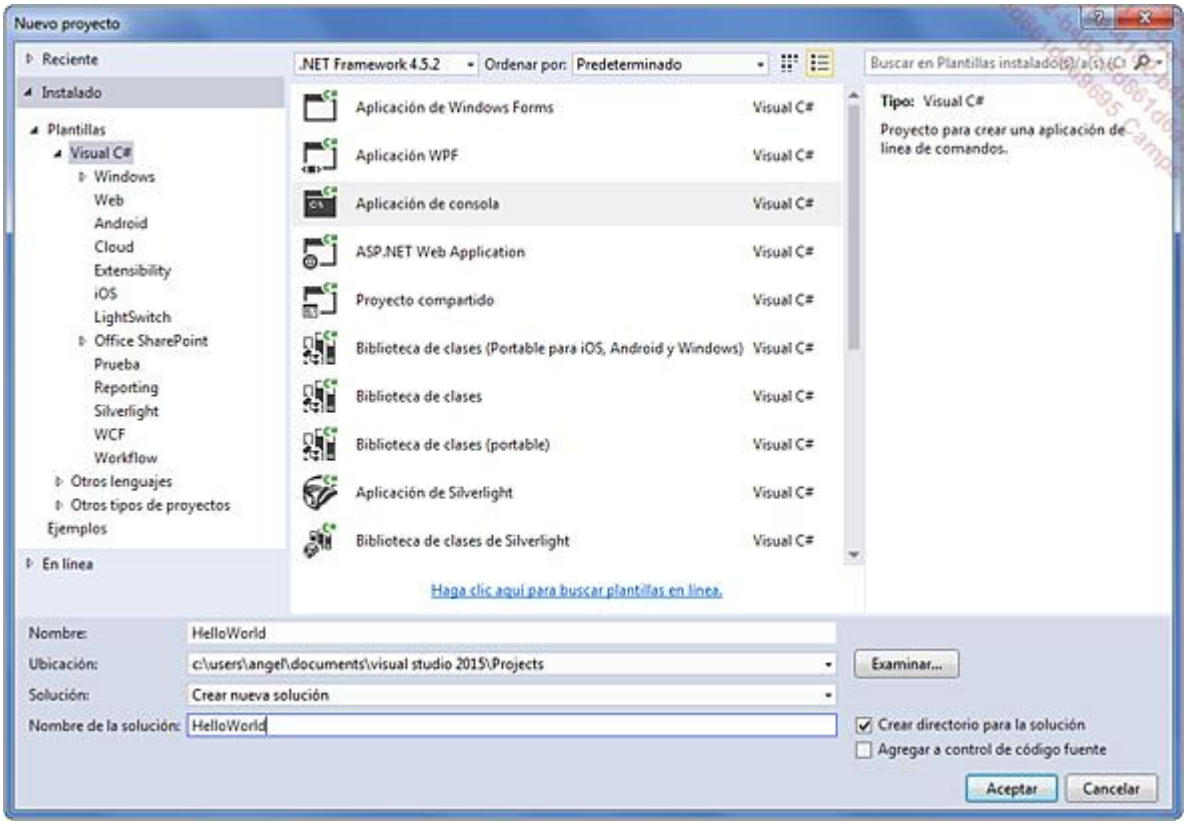
Vaya a la opción de menú **Archivo - Nuevo - Proyecto**.



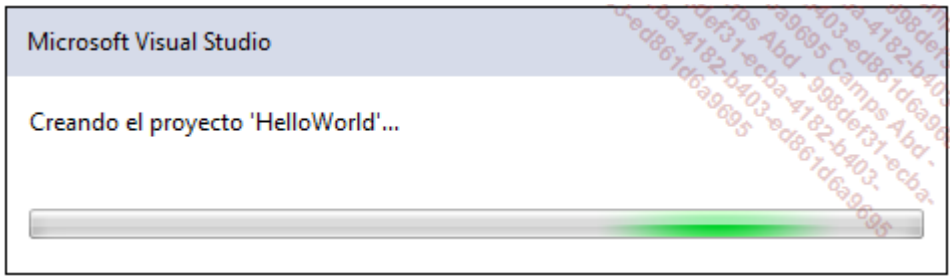
Esta acción ejecuta el asistente de creación de aplicaciones. Esta asistente genera el marco de la futura aplicación, en función de sus opciones iniciales.

Entre las plantillas, seleccione la familia **Visual C# Windows** y la plantilla **Aplicación de consola**, que es la forma la más sencilla para hacer nuestras primeras pruebas. Asigne el nombre HelloWorld al proyecto, seleccione un directorio de trabajo y después asigne también el nombre HelloWorld a la solución. Veremos que una aplicación normalmente no es

monolítica. Puede estar formada por varios componentes (por ejemplo, varias DLL y un EXE). La solución de Visual Studio le permite crear este conjunto de componentes, cada componente construido a partir de un proyecto.



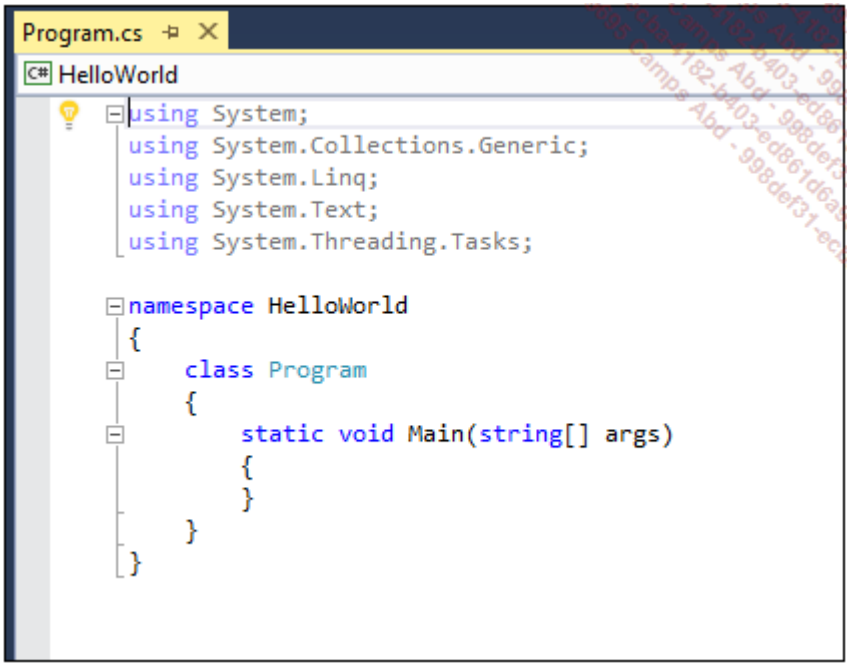
Una vez que se validan los argumentos, el asistente prepara el proyecto.



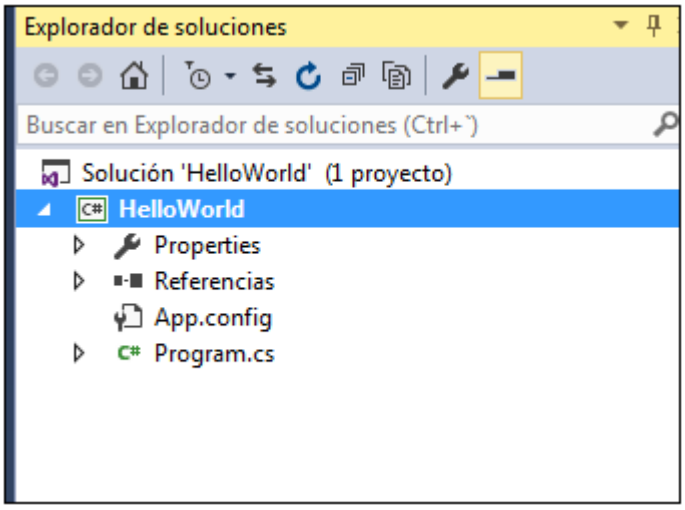
Vista nuestro Facebook:

https://www.facebook.com/dogr4mcode.web

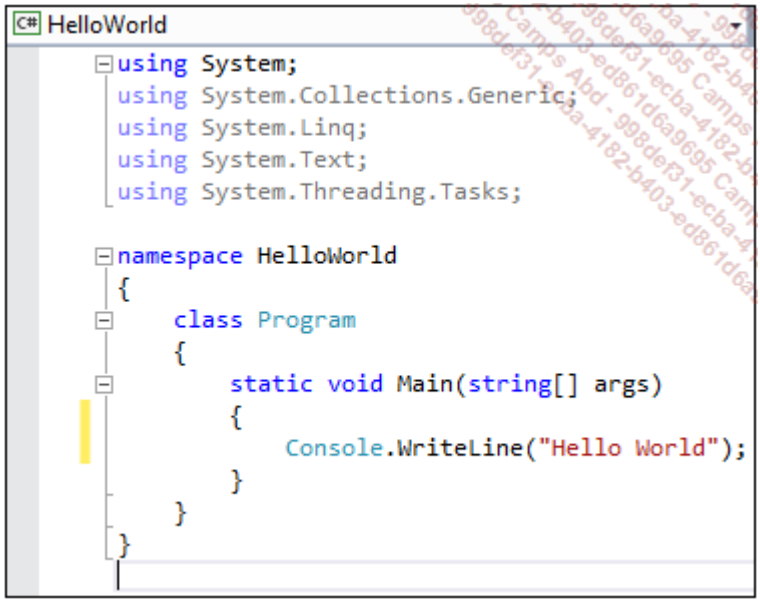
Visual Studio muestra en la parte derecha el contenido del archivo de código fuente de nuestra solución: **Program.cs**. Se utilizada la extensión cs (por C Sharp) para estos archivos de texto, que contiene el código de la aplicación.



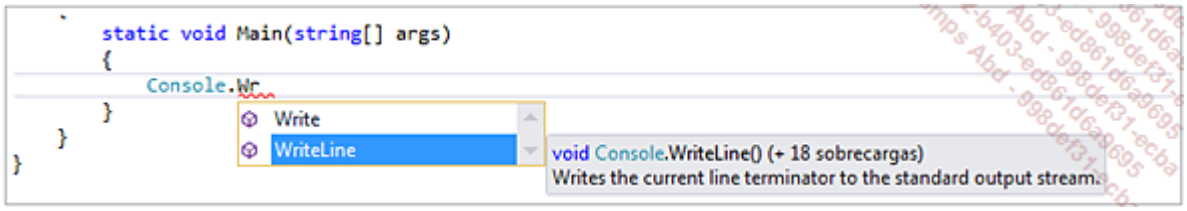
La parte izquierda muestra el **Explorador de soluciones** con su único proyecto **HelloWorld**:



Complete el contenido de Program.cs como sigue:

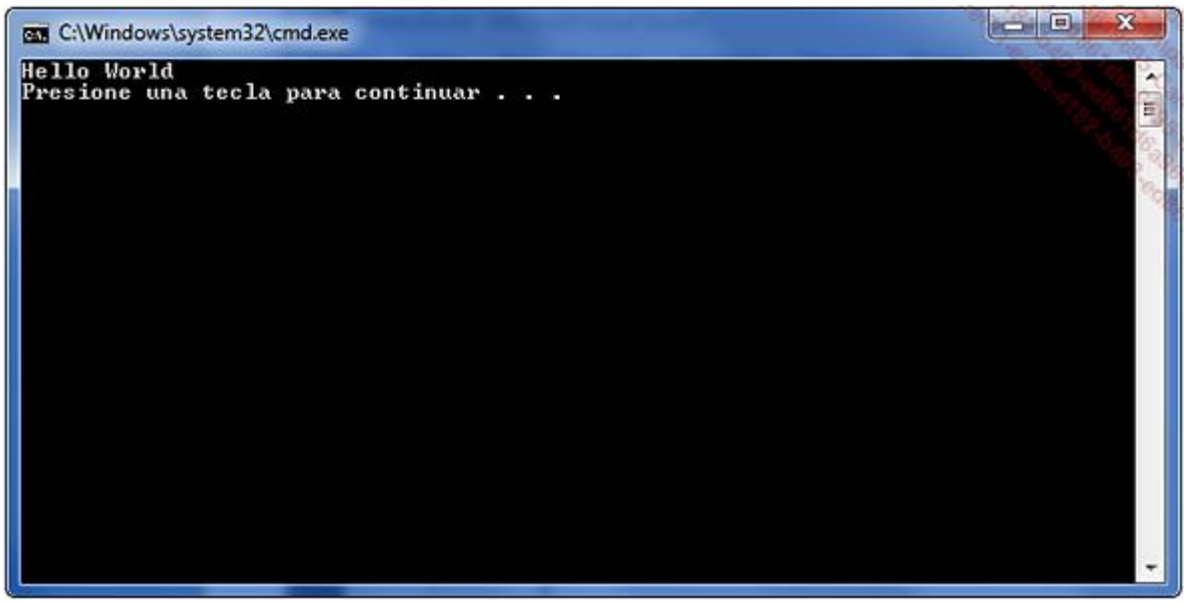


Sin duda habrá observado que durante la redacción de esta línea, se muestra una ayuda contextual. Esta funcionalidad se llama IntelliSense y rápidamente comprobará que será su aliada durante las fases de escritura de su código.



Capaz de "completar" su entrada de texto, también ofrece ayuda acerca del uso del método.

Ejecute el programa con [Ctrl][F5].



Objetivo conseguido: se muestra la ventana de la consola con la cadena "Hello World".

"En C#, todo está tipado"

El término genérico "**tipo**" agrupa las clases, estructuras, interfaces, enumeraciones y delegados. Estos cinco tipos se describen en la CTS (Common Type System) para que los compiladores de diferentes lenguajes puedan generar código explotable por la CLR (Common Language Runtime). Un programa utiliza diferentes tipos y un ensamblado puede implementar varios tipos.

A continuación se muestran las definiciones resumidas de los diferentes tipos propuestos por C#:

- El tipo "Clase" es la implementación C# de lo que se ha presentado en los primeros capítulos. La clase evidentemente es el tipo más utilizado en las aplicaciones. El capítulo Creación de clases define de manera precisa la sintaxis de declaración, asignación y uso.
- El tipo "Estructura" es una herencia del lenguaje C. Antes de la democratización de la programación orientada a objetos, las estructuras eran el medio más común para los desarrolladores para construir sus propios tipos. De momento, nos quedamos con que las estructuras de C# son muy cercanas a las clases y que, cuando se utilizan sabiamente, permiten mejorar el rendimiento de una aplicación. Veremos en el siguiente capítulo que el framework .NET integra la mayor parte de sus tipos "sencillos" en las estructuras. Sepa que las estructuras no existen en Java.
- El tipo "Interfaz" es muy utilizado en el framework .NET y contribuye a la comunicación entre las clases. Por el momento nos quedamos con que una interfaz es una clase sin código que formaliza un lote de métodos obligatorios para la clase que la implementa. El capítulo Herencia y polimorfismo trata sobre este asunto.
- El tipo "Enumeración" permite la definición de listas clave-valor y la creación de datos cuyos contenidos se limitarán a estas claves. Por ejemplo, se puede crear un tipo Día que puede contener de lunes a domingo. Si durante la escritura del programa se intenta copiar en un objeto de este tipo la clave Marzo, se producirá un error de compilación. En C#, este tipo aporta un lote de métodos que permite gestionar esta lista por programación.
- El tipo "Delegate" (Delegado) encapsula la noción de puntero de función de C/C++, origen de buena parte de los problemas, empezando por asignarle un tipado fuerte. De hecho, el puntero de función "convencional" no es otra cosa que una dirección de memoria sin ninguna otra precisión sobre la firma, de modo que la aplicación se detiene con un error cuando los argumentos que se pasan no se corresponden con los argumentos esperados. Por este motivo el tipo delegate de C# se va a definir de manera precisa con la firma del método que se le asocia. Seguidamente, la instancia de tipo delegate, generalmente creada dentro de una clase que establece la comunicación con otras, gestiona una lista "de abonados" a través de una sintaxis desconcertante por su simplicidad. De hecho, basta con utilizar el operador += del delegate para registrarse como abonado a la lista de difusión y -= para eliminarse de la misma. Los delegate se utilizan mucho en C#; los encontraremos

habitualmente en las interfaces gráficas para que los componentes puedan notificar a la aplicación sus cambios de estado.

Durante este capítulo se abordarán nociones ilustradas por fragmentos de código. Estos fragmentos de código utilizan las sintaxis descritas en los capítulos siguientes, pero la comprensión de los capítulos siguientes pasa por presentar ahora estos ejemplos. Así que, de momento, hay que aceptar sin cuestionarse la sintaxis de los ejemplos, que veremos en profundidad más adelante.

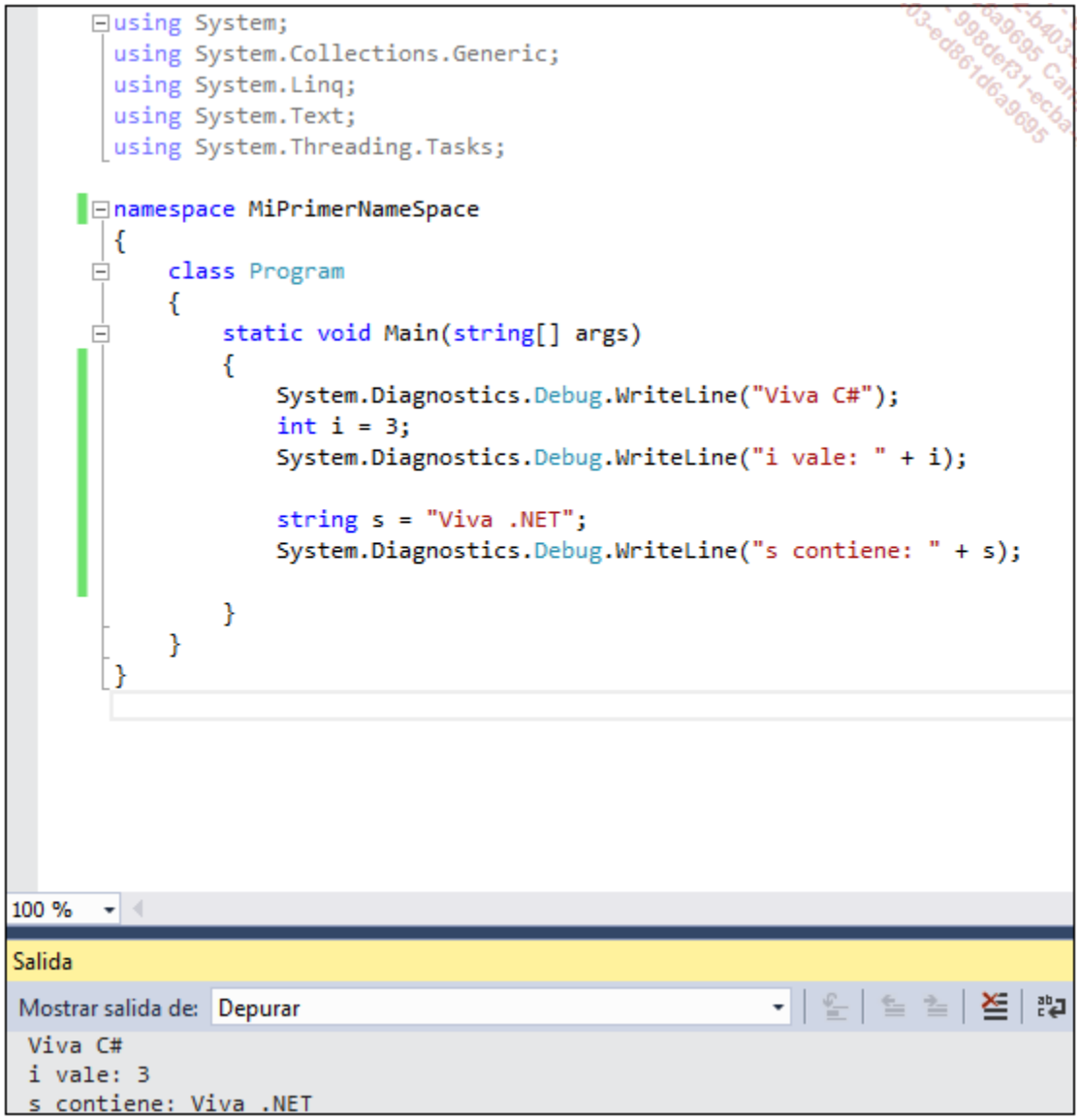
Introducción a System.Diagnostics.Debug

Estos ejemplos utilizan clases de prueba y también una clase de sistema, llamada System.Diagnostics.Debug. Esta clase permite, entre otras cosas, escribir mensajes en la ventana **Salida** de Visual Studio y también comprobar las condiciones que se pasan como argumentos.

Sintaxis de visualización en la ventana de Salida de Visual Studio:

```
System.Diagnostics.Debug.WriteLine("el mensaje");
```

Ejemplo de formas de uso sencillas y compuestas:



El método System.Diagnostics.Debug.Assert permite comprobar si una condición es verdadera durante la ejecución de su código. Utilizando este método, usted no interviene en la ejecución del programa como tal, sino que comprueba que lo previsto hasta un punto determinado del código es correcto. Si la condición es falsa, se mostrará un cuadro de diálogo para informarle.

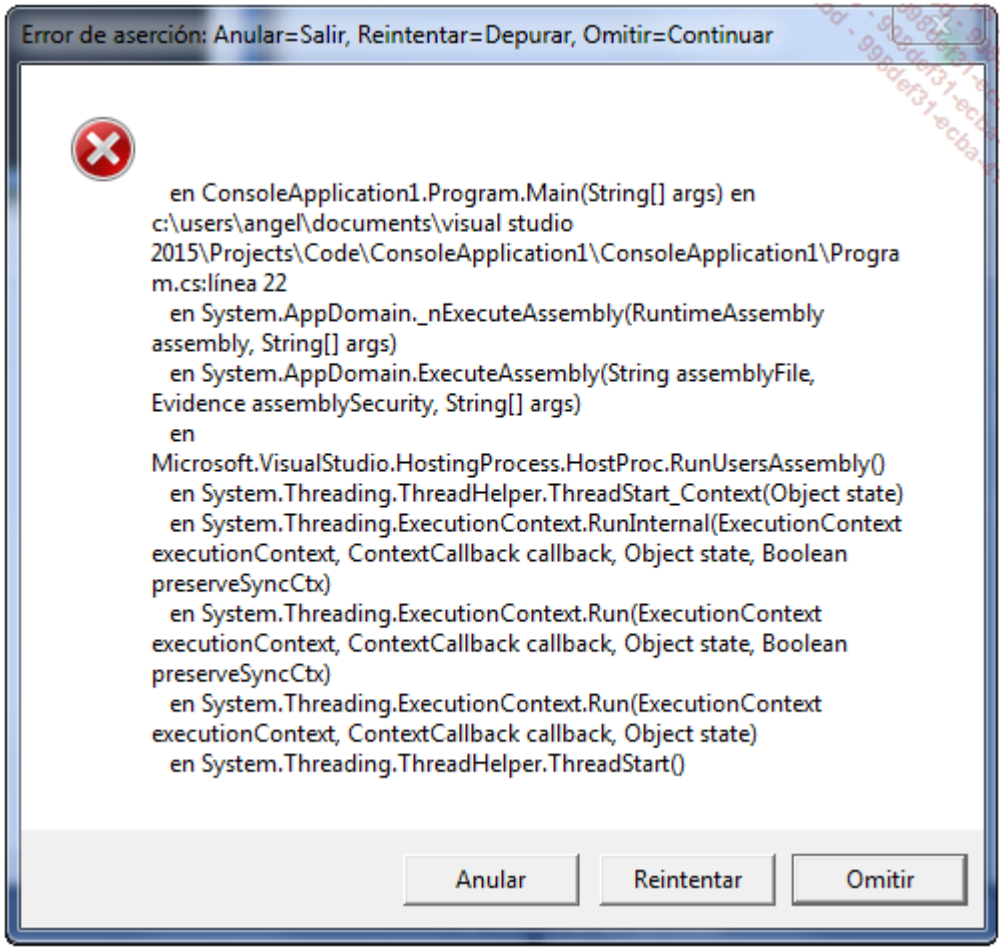
Sintaxis de uso del método System.Diagnostics.Debug.Assert:

```
System.Diagnostics.Debug.Assert(<condición>);
```

Ejemplos de uso del método Assert

```
    // Comprobación de la condición "1 es diferente de 2"
    System.Diagnostics.Debug.Assert(1 != 2);
    // Como la condición es verdadera, el programa
    // pasa a la línea siguiente



    // Para visualizar el resultado del comando
    // cuando una condición no se verifica,
    // se "fuerza" un error en la línea siguiente
    System.Diagnostics.Debug.Assert(1 == 2);
```

Durante la ejecución de la segunda línea del fragmento, el programa muestra un cuadro de diálogo y espera a que el usuario lo cierre antes de proseguir la ejecución del código.



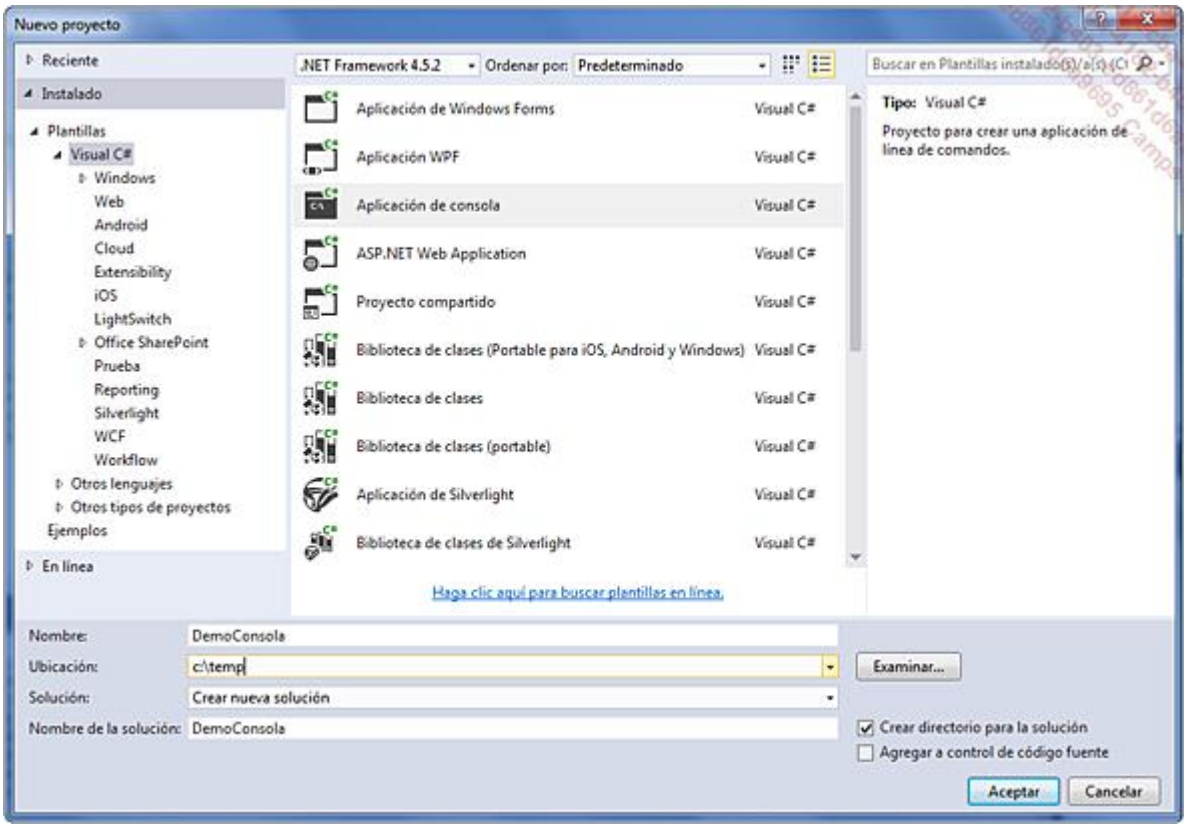
De esta manera, puede comprobar si lo que ha previsto se realiza correctamente. Se utiliza System.Diagnostics.Debug.Assert principalmente durante las fases de puesta a punto

para eventualmente añadir código de protección. Veremos más adelante que Visual Studio genera una versión de "puesta a punto" (Debug) y una versión "producción" (Release). System.Diagnostics.Debug.Assert no tiene ningún efecto en un código compilado en modo "producción".

Introducción a System.Console

Ya se ha utilizado en el capítulo Introducción al framework .NET y a VS, para mostrar el clásico Hello World en pantalla; la consola va a servir de soporte a varios ejemplos a continuación. Este entorno de ejecución, muy resumido, presenta la ventaja de poder mostrar cadenas en pantalla y leer las entradas por teclado de una manera sencilla.

Para la creación del proyecto, se elige el tipo **Aplicación de consola**.



A continuación se muestran los principales comandos que se utilizaran:

Mostrar una cadena seguida de un cambio de línea:

```
Console.WriteLine("Mensaje a mostrar...");
```

Mostrar un tipo primitivo sin cambio de línea:

```
int i = 358;
Console.Write(i);
```

Lectura de una cadena de caracteres introducida por teclado y finalizada por la tecla [Intro]:

```
string introduccionTeclado = Console.ReadLine();
```

"Todo el mundo hereda de System.Object"

El tipo System.Object es la base directa o indirecta de todos los tipos de .NET, tanto de los existentes como de los que se crean nuevos (la noción de herencia ya se ha abordado en los primeros capítulos). La herencia de Object es implícita y, por tanto, su declaración es inútil. Todos los tipos heredan de sus métodos e incluso pueden sustituir algunos.

Esto es lo que hace System.ValueType que, en la herencia de tipos de .NET, se convierte en la base de la familia "Valores", adaptándose a los métodos de System.Object.

1. Los tipos Valores

La familia "Valores" se divide en dos partes:

- Las enumeraciones
- Las estructuras

A su vez, las estructuras se dividen en:

- Tipos numéricos:
- Los tipos enteros:
- Los tipos de coma flotante:

Tipo	Tamaño
sbyte	Entero con signo de 8 bits
byte	Entero sin signo de 8 bits
char	Carácter UNICODE 16 bits
short	Entero con signo de 16 bits
ushort	Entero sin signo de 16 bits
int	Entero con signo de 32 bits
uint	Entero sin signo de 32 bits
long	Entero con signo de 64 bits
ulong	Entero sin signo de 64 bits

Tipo	Precisión
float double	7 dígitos 15-16 dígitos

1. El tipo decimal (adaptado a los cálculos financieros):

- Tipo booleano: tipo bool.
- Estructuras de usuario (son las que usted escribe). Observe que es el único medio de construir tipos Valores, porque la herencia del tipo System.ValueType está prohibida.

Tipo	Precisión
decimal	28-29 dígitos

De hecho, los tipos "sencillos" enunciados en las tres tablas son alias a las estructuras del espacio System. Así hablamos de tipos valores integrados. Por tanto, se pueden declarar:

- Indicando el nombre completo de la estructura (ejemplo: System.Int32).
- Utilizando el alias que tiene asociado en la gramática de C# (ejemplo: int).

Un tipo Valor se debe inicializar antes de utilizarlo. De lo contrario, se producirá un error de compilación. El método más clásico consiste en continuar la definición con el signo de asignación y su valor inicial. También es posible instanciar un tipo Valor con ayuda de la palabra clave new. En este caso, se pasará por el constructor (se verá más adelante) por defecto del tipo. El constructor ajusta el valor a 0 para los tipos numéricos y a false para el tipo booleano.

```
// Declaración poco utilizada
System.Int32 j = 5;


// Declaración clásica
int i = 5;


// Se pasa por el constructor
char c = new char();
// c vale ahora \0
```

Los tipos Valor se almacenan en una zona de memoria llamada pila (stack). Son rápidos de crear, pero tienen un tiempo de vida limitado a una determinada sección (generalmente las llaves que lo rodean). Se destruyen tan pronto como la ejecución del programa sobrepasa los límites de su definición. Por ejemplo, un entero definido como variable local de un método se destruirá tan pronto como el método termine.

```
public void Calcular(int i)
{
    int j = 0;
    //...
}   // Después de esta llave, i y j se destruyen
```

La copia de un objeto de tipo Valor en otro objeto implica la copia del contenido (y no del contenedor como sucede con el tipo Referencia, que se verá más adelante).

Ejemplo:

```
public void Calcular()
{
    int i = 3;
    int j = i;

    j = j + 2;
    // Aquí j vale 5 e i vale 3
}
```

Es imposible heredar de un tipo Valor. Por el contrario, una estructura puede heredar de una interfaz y, por tanto, implementarla.

Un tipo Valor obligatoriamente debe tener un valor salvo si ha recibido el atributo nullable. En este caso, puede tomar todos los valores para los que fue previsto y, además, tomar el valor null.

Ejemplo:

```
bool b1 = true;
// b1 puede contener true o false
bool? b2 = null;
// b2 puede contener true false o null
```

Es particularmente importante entender el sentido del valor null. Significa que el valor del objeto es indefinido. El ejemplo clásico es un registro parcial en una base de datos. De hecho, es muy frecuente que no se conozcan todos los campos durante la creación de un registro. Por ejemplo, la media final anual de un estudiante no se podrá rellenar durante la creación de su ficha de inscripción. En este caso, durante la definición del campo de la tabla, el lenguaje SQL (lenguaje para bases de datos) ofrece la posibilidad de utilizar la restricción NULL para autorizar el registro "vacío". Encontramos esta noción en C#. Los tipos "nullables" son instancias de la estructura System.Nullable<T>, donde T es un tipo valor (volveremos más adelante sobre esta noción de generalidad, consistente en asignar un tipo como argumento). La sintaxis T? es un acceso directo para System.Nullable<T>.

2. Los tipos Referencia

La familia Referencia agrupa las clases, los delegados y las interfaces.

Al contrario que los tipos Valor, los tipos Referencia almacenan las referencias a los datos. Estos datos se almacenan en una zona de memoria llamada el heap. Son más largas de crear que en el stack, pero son accesibles desde otras instancias de clase. Su ciclo de vida termina cuando ya no la utilicemos. Mientras haya al menos una referencia activa en la zona de datos, se mantiene. Tan pronto como no haya más referencias, la zona se considera como no utilizada y su destrucción se gestionar por el garbage collector.

Un tipo Referencia puede no hacer referencia a nada (o no todavía). En este caso, contiene null. Es inútil definir una referencia como nullable, porque autoriza el valor null de manera nativa.

La instanciación de una clase se realiza solo con la palabra clave new.

Ejemplo:

```
    public void Tratamiento()
    {
        MiClase miclase;
        // aquí miclase vale null
        //...
        miclase = new MiClase();
        // aquí miclase referencia a una
        // instancia de MiClase
    }
    // después de esta llave, la referencia miclase no se
    // ha "transportado" a ninguna otra
    // instancia, el objeto que referencia
    // será ilegible cuando se borre
```

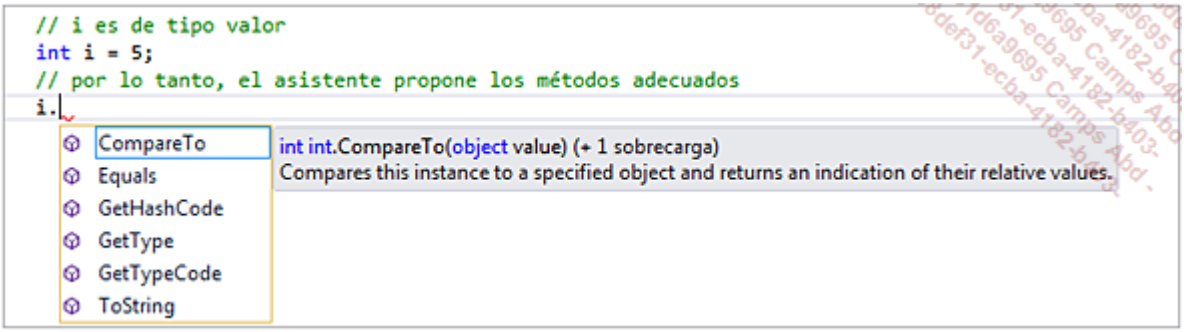
Una aplicación puede tener varias referencias a la misma instancia de una clase.

```
public void Tratamiento()
{
    MiClase mc1 = new MiClase();
    MiClase mc2 = mc1;

    // mc1 y mc2 referencian al mismo objeto en el heap
}
```

En este fragmento, cualquier acción realizada desde la referencia mc1 tendrá el mismo efecto en el objeto como si se hubiera realizado desde mc2. Por tanto, las referencias al stack se pueden "intercambiar" entre diferentes métodos, mientras el objeto en el heap sigue siendo el mismo.

3. Boxing - Unboxing

Un tipo Valor contiene un valor y, por tanto, no es un objeto con métodos. Aun así, hemos visto que todos los tipos heredan de System.Object, que ofrece un juego de métodos básicos que vamos a comentar muy pronto. Para permitir esta conversión de valor a referencia, la CLR utiliza un proceso llamado boxing. De hecho, la CLR crea un objeto en el heap que contiene el valor del tipo fuente y que ofrece los métodos de sus clases de base. La siguiente captura muestra un boxing "implícito", que se ejecutará cuando queramos utilizar los métodos de System.Object directamente, a partir de un entero.



Puede ser que queramos conservar el efecto boxing más tiempo. En este caso, se codifica un boxing "explícito", como el que se muestra en el siguiente fragmento de código.

```
int i = 5;
// Boxing explícito
object o = i;
//... resto del código que utiliza o
```

La operación inversa, unboxing, realiza la copia de un valor contenido en un objeto a un tipo Valor correspondiente. El desarrollador utiliza el operador de cast para "unboxar" el valor, pero

al contrario que en C/C++, la CLR va a comprobar que el cast ((int) en el siguiente fragmento) corresponde al tipo apropiado.

```
    int i = 5;


    // Boxing explícito
    object o = i;
    //... resto del código que utiliza o


    // Unboxing
    int j = (int)o;
```

4. Utilización de los métodos de System.Object

Ahora es momento de descubrir algunos métodos de la clase raíz de la jerarquía de tipos .NET, a saber System.Object.

```
public class Object
{
  // Métodos virtuales
  public virtual bool Equals(object obj);
  public virtual int GetHashCode();
  public virtual string ToString();
  protected virtual void Finalize();
  // Métodos no virtuales
  public Type GetType();
  protected object MemberwiseClone();
  // Miembros de tipo static
  public static bool Equals(object objA, object objB);
  public static bool ReferenceEquals(object objA, object objB);
}
```

Este párrafo va a abordar las nociones de herencia presentadas globalmente en los primeros capítulos, pero que se estudiarán de manera mucho más precisa en un próximo capítulo.

El tipo object (todo en minúscula) es un sinónimo del tipo System.Object.

a. Equals

```
public virtual bool Equals(object obj);
```

Este método permite comparar dos objetos.

El comportamiento de la prueba depende de los tipos comprobados.

- Si son de tipo Valor entonces se compararán valores.
- Si son de tipo Referencia entonces se compararán las referencias a los objetos en el heap.

Ejemplo:

```
            int a = 3;
            int b = 4;
            // La prueba se hace entre dos tipos Valor
            // por lo que la comparación se hace entre los... valores
            if (a.Equals(b) == true)
            {
                // Nunca se llegará a esta línea
                // porque a vale 3 y b vale 4
            }
            MiClase mc = new MiClase();
            MiClase mc2 = mc;
            // La prueba se hace entre dos tipos Referencia
            // por lo que se comparan las referencias
            if (mc.Equals(mc2) == true)
            {
                // Se llegará a esta línea
                // porque mc y mc2 referencian al mismo objeto
            }
```

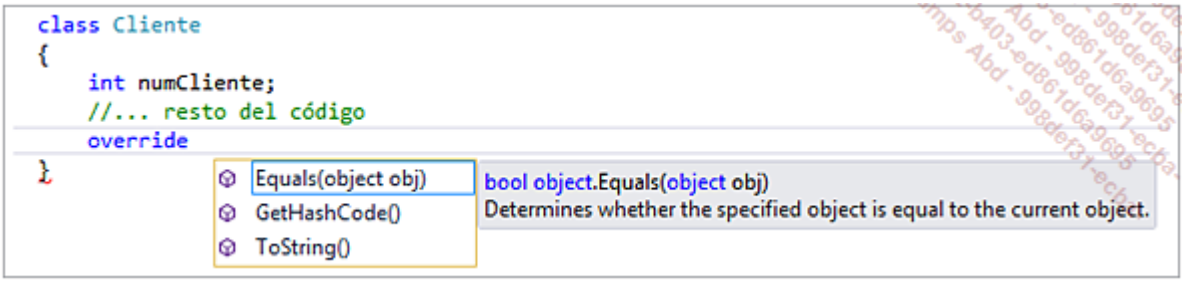
Cuando escribe sus propias clases puede redefinir el comportamiento de este método para que implemente pruebas adaptadas a la comparación de dos instancias. Por ejemplo, si tiene

una clase Cliente con un identificador único, puede comparar este identificador para saber si ambas instancias tienen la misma "correspondencia".

El asistente de Visual Studio le ayuda a "rescribir" a su manera los métodos de la clase de base de su objeto. Por ejemplo, tiene esta clase Cliente para completar con un método Equals:

```
class Cliente
{
    int numCliente;
    // ... resto del código
}
```

Escriba la palabra clave override seguida del carácter espacio, dentro de la definición de la clase.



En ese momento, el asistente muestra todos los métodos que se pueden sustituir (ver el capítulo relativo a la herencia para obtener más información sobre la sustitución).

Seleccione el método Equals en la lista.

```
class Cliente
{
    int numCliente;
    // ... resto del código
    public override bool Equals(object obj)
    {
        return base.Equals(obj);
    }
}
```

El fragmento anterior muestra el código autogenerado; por el momento, es suficiente con llamar a la operación de base. Como Cliente es de tipo referencia, la operación básica se hará solo para la comparación de las referencias de los dos objetos. Esto no es lo que queremos para nuestro tipo Cliente, por lo que vamos a modificar el cuerpo del método Equals.

Observe también que el argumento del método Equals es de tipo object y no Cliente. Esto no es grave, porque Cliente hereda de object y por tanto, en virtud del polimorfismo, se puede considerar un Cliente como un tipo de object. Por tanto, será normal pasar como argumento a Equals una referencia a un objeto Cliente.

En el cuerpo mismo del método Equals será necesario comprobar que el argumento obj es de tipo Cliente antes de poder comparar los dos campos numCliente. Aquí entran en escena los operadores is y as.

```
class Cliente
{
    int numCliente;
    // ... resto del código
    public override bool Equals(object obj)
    {
        // ¿Es obj de tipo Cliente?
        if (obj is Cliente)
        {
            // Sí. Se crea una referencia temporal
            // esta vez "fuertemente tipada Cliente"
            Cliente c = obj as Cliente;
            // Ahora es sencillo comparar los dos campos.
            return numCliente == c.numCliente;
        }
        // En el resto de casos,
        // la comprobación de igualdad es falsa
        return false;
    }
}
```

El operador is permite saber dinámicamente si un objeto concreto es de un determinado tipo.

El operador as permite considerar un objeto como si fuera de un determinado tipo. Es posible utilizar as directamente sin la llamada previa a is. Si el tipo utilizado no se puede considerar como el tipo deseado, entonces la referencia devuelta por as será null. Es la diferencia principal con un operador de cast clásico que, en el mismo caso, provocaría una excepción.

En el ejemplo anterior, cuando is comprueba que el objeto es de tipo Cliente, as se puede sustituir por un cast clásico:

```
    Cliente c = (Cliente)obj;
```

El método Equals tiene dos formas diferentes: un método instanciado y uno de tipo static. El capítulo Creación de clases presenta la utilidad de los métodos de tipo static.

```
    Cliente c1 = new Cliente();
    Cliente c2 = new Cliente();
    if (c1.Equals(c2))
    { //...
    }
    // Es equivalente a:
    if (Object.Equals(c1, c2))
    { //...
    }
```

b. GetHashCode

```
public virtual int GetHashCode();
```

En programación orientada a objetos es importante que cada instancia pueda tener una referencia única. El método GetHashCode se puede utilizar para esto y, de esta manera, llamarse durante la construcción de listas de referencias.

Si GetHashCode se utiliza con un tipo Valor, entonces devolverá el valor.

Si GetHashCode se utiliza con un tipo Referencia, entonces devolverá la referencia al objeto.

```
            int i = 3;
            int j = 3;
            int hashOfi = i.GetHashCode();
            int hashOfj = j.GetHashCode();
            // Los HashCodes de i y de j valen 3

            Cliente c3 = new Cliente();
            Cliente c1 = new Cliente();
            int hashOfC1 = c1.GetHashCode();
            Cliente c2 = new Cliente();
            int hashOfC2 = c2.GetHashCode();
            // Los HashCodes de c1 y de c2 son
            // totalmente diferentes
            // (ex: 45653674 y 41149443)
```

Es libre de redefinir el comportamiento de este método para una de sus clases.

```
    class Cliente
    {
        int numCliente = 10;
        // ... resto del código
        public override bool Equals(object obj)
        {
            // ¿Es obj de tipo Cliente?
            if (obj is Cliente)
            {
                // Sí. Se crea una referencia temporal
```

```
                // esta vez "fuertemente tipada Cliente"
                Cliente c = obj as Cliente;
                // Ahora es sencillo comparar los dos campos.
                return numCliente == c.numCliente;
            }
            // En el resto de casos,
            // la comprobación de igualdad es falsa
            return false;
        }
        public override int GetHashCode()
        {
             // En la instancia Cliente, es el atributo numCliente
            // el que especifica el Cliente de manera única
            return numCliente;
        }
    }
```

GetHashCode se utiliza principalmente para manipular rápidamente objetos en las colecciones de tipo diccionario.

Cuando la llamada al método Equals entre dos objetos devuelve true, los valores devueltos por sus respectivos GetHashCode deben ser idénticos.

Si dos objetos devuelven un mismo valor para GetHashCode, no significa que Equals los considere idénticos. Es el caso extremo de las "colisiones", para el que el algoritmo de cálculo ha generado varias veces el mismo hash code.

Cuando se sobrecargan los métodos Equals y GetHashCode, también se suele sobrecargar los operadores == y !=, que se presentan más adelante.

c. ToString

```
public virtual string ToString();
```

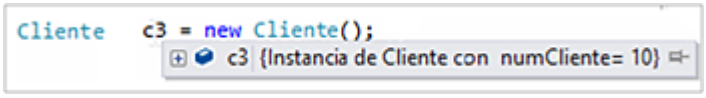
Es muy útil durante las fases de puesta a punto. El método ToString permite devolver, en forma de cadena de caracteres, información resumida de la instancia de la clase. Puede poner en esta información lo que le parezca más importante para determinar la fuente de un funcionamiento incorrecto.

En el siguiente ejemplo de código ToString devuelve el contenido del campo numCliente.

```
    class Cliente
    {
        int numCliente = 10;
        // ... resto del código
        public override bool Equals(object obj)
        {
            // ¿Es obj de tipo Cliente?
            if (obj is Cliente)
            {
                // Sí. Se crea una referencia temporal
                // esta vez "fuertemente tipada Cliente"
                Cliente c = obj as Cliente;
                // Ahora es sencillo comparar los dos campos.
                return numCliente == c.numCliente;
            }
            // En el resto de casos,
            // la comprobación de igualdad es falsa
            return false;
        }

        public override int GetHashCode()
        {
            // En la instancia Cliente, es numCliente el
            // que especifica el Cliente de manera única
            return numCliente;
        }
```

```
        public override string ToString()
        {
            return "Instancia de Cliente con numCliente = " + numCliente;
        }
    }
```

La explotación del resultado de ToString depende de la estrategia seleccionada para la puesta a punto de su clase. Es posible mostrarlo en la ventana **Salida** de Visual Studio o incluso registrarlo en un archivo de trazas. Sin embargo, un uso inmediato y muy práctico es para el unlocker de Visual Studio, que permite ejecutar el programa línea a línea para comprobar el funcionamiento. En este modo, cuando se sitúa el cursor sobre un objeto, el unlocker muestra una ventana que contiene el resultado de la llamada a ToString del objeto subyacente.



d. Finalize

El método Finalize está relacionado con el proceso de destrucción de los objetos. El capítulo Creación de clases contiene un párrafo que presenta el mecanismo de excepciones en .NET. Por el momento, es suficiente con recordar que el garbage collector invocará a este método cuando un objeto quede no accesible, es decir, cuando no haya ninguna referencia en el programa que lo ha creado, justo antes de liberar la memoria.

En C#, el método Finalize aparece de la siguiente forma:

```
~NombreDeLaClase()
{
   // Código de 'limpieza'
}
```

En C# los Finalize de las clases heredadas llaman automáticamente a los Finalize de las clases de base.

Volveremos a este método un poco más tarde.

e. Object.GetType y los operadores typeof e is

El polimorfismo es un potente sistema que permite considerar objetos como pertenecientes a una misma familia de manera uniforme. Dicho esto, puede ser útil conocer el tipo "nativo" de una instancia.

En el siguiente ejemplo, una clase MantenimientoVehiculo ofrece un método Mantenimiento, que recibe como argumento un objeto de tipo Vehiculo. Se puede utilizar con cualquier objeto que herede de la clase Vehiculo, como VehiculoAmotor y VehiculoApedales. El método necesita conocer el tipo real para adaptar la operación.

Es interesante utilizar el método GetType sobre el objeto porque su resultado se podrá comparar con el retorno de un typeof, que se utiliza directamente sobre un tipo, como aquí:

```
class Vehiculo
 { /*...*/ }


 class VehiculoAmotor: Vehiculo
 { /*...*/ }


 class VehiculoApedales: Vehiculo
 { /*...*/ }


 class MantenimientoVehiculo
 {
     public void Mantenimiento(Vehiculo v)
     {
         if (v.GetType() == typeof(VehiculoAmotor) )
         {
             //...
         }
     }
 }
```

Antes se ha presentado el operador is, que forma parte de lo mismo. A continuación se muestran la diferencias entre ambos.

- Si el objeto es resultado de una herencia de clases, el tipo devuelto por GetType será su tipo exacto (el más especializado en la herencia).
- El operador is permite probar si el objeto es de un tipo concreto, la prueba será verdadera para todos los tipos y subtipos que formen la herencia.

- GetType devuelve un objeto de tipo Type que hay que comparar con el resultado de typeof.
- is se utiliza directamente entre un objeto y el nombre de la clase, la prueba devuelve un booleano.

f. Object.ReferenceEquals

Este método static (razón por la que Object. le precede) de la clase Object permite saber si dos objetos hacen referencia a la misma entidad del heap. Este método no se puede sustituir durante una herencia. No tiene ningún interés para los tipos Valor, para los que siempre devuelve false.

```
public static bool ReferenceEquals (
   Object objA,
   Object objB
)
```

Ejemplo:

```
    Cliente c1 = new Cliente();
    Cliente c2 = new Cliente();
    Cliente c3 = c2;
    if (Object.ReferenceEquals(c1, c2))
    {
        // Nunca se llegará a esta línea
        // porque c1 y c2 son dos referencias diferentes
    }
    if (Object.ReferenceEquals(c2, c3))
    {
        // Se llegará a esta línea
        // porque c2 y c3 hacen referencia al mismo objeto
    }
```

Este método también es de tipo static. Volveremos sobre esta sintaxis particular un poco más tarde. De momento, es suficiente con recordar que un método de tipo static no pertenece a una instancia dada, sino a su tipo. Esta es la razón por la que recibe como argumento las dos instancias a comparar.

g. Object.MemberwiseClone

Este método realiza una copia "parcial" del objeto actual.

```
protected Object MemberwiseClone ()
```

¿Por qué "parcial"?

En primer lugar, este método solo funciona con objetos de tipo Referencia. Debido a su acceso de tipo protected, solo se puede llamar por una instancia heredada. Esta clase heredada ofrecerá un método, de tipo public, que se debe utilizar para obtener su clonado. La interfaz ICloneable generalmente se utiliza para normalizar este método. El siguiente código muestra este tipo de implementación para una clase Cliente.

Observe que, por comodidad, la regla de la encapsulación no se respeta en los siguientes fragmentos de código.

A continuación se muestra un fragmento de código que implementa la interfaz:

```
    class Cliente: ICloneable
    {
        public int numCliente = 10;


        public object Clone()
        {
            return MemberwiseClone();
        }
            //...
    }
```

Para terminar, a continuación se muestra un fragmento de código que solicita un clonado de la clase Cliente:

```
    Cliente clienteOriginal = new Cliente();
    //...
    Cliente clienteClonado = (Cliente)clienteOriginal.Clone();
```

Observe que el método Clone devuelve un objeto de tipo object y que el usuario debe "transformar" este retorno en tipo Cliente, es decir, forzar al compilador a considerarlo como un tipo Cliente.

Por tanto, el método protegido MemberwiseClone genera un segundo objeto basado en el primero como modelo..

Para los miembros de tipo Valor de la clase a clonar, el método duplica los contenidos, realizando una copia muy precisa (bit a bit). El miembro clonado tiene el mismo valor que el miembro original y, a continuación, ambos miembros no vuelven a tener ninguna relación. Cualquier modificación de un miembro de tipo Valor del lado del clon, no tiene ninguna incidencia en el original, como se demuestra con el siguiente fragmento de código:

```
    Cliente clienteOriginal = new Cliente();
    Console.WriteLine("Inicial:" + clienteOriginal.numCliente);

    Cliente clienteClonado = (Cliente)clienteOriginal.Clone();
    Console.WriteLine("Clone:" + clienteClonado.numCliente);
    clienteClonado.numCliente = 20;
    Console.WriteLine("Clon:" + clienteClonado.numCliente);

    Console.WriteLine("Inicial:" + clienteOriginal.numCliente);
```

Muestra en la consola:

```
Inicial:10
Clon:10
Clon:20
Inicial:10
```

Para los miembros de tipo Referencia, los contenidos también se van a duplicar, pero finalmente siempre harán referencia a los mismos objetos.

Los objetos referenciados no se clonan en la operación, lo que justifica el calificativo de copia "parcial".

Cualquier modificación realizada a un miembro de tipo referencia del objeto clonado tendrá incidencia en el original.

Para ilustrar este principio, volvamos a nuestra clase Cliente y añadamos un miembro referencia a una clase de tipo Empresa.

```
class Empresa
{
    public string nombre;
}

class Cliente: ICloneable
{
    public Empresa empresa = new Empresa();

    public int numCliente = 10;

    public object Clone()
    {
        return MemberwiseClone();
    }

    // ... resto del código
}
```

Ahora, realicemos una modificación a partir del miembro empresa de la instancia clonada para comprobar su repercusión en la instancia original:

```
Cliente clienteOriginal = new Cliente();
clienteOriginal.empresa.nombre = "Empresa Original";
Console.WriteLine("Inicial: " + clienteOriginal.empresa.nombre);

Cliente clienteClonado = (Cliente)clienteOriginal.Clone();

Console.WriteLine("Clon: " + clienteClonado.empresa.nombre);
clienteClonado.empresa.nombre = "Empresa Clonada";
Console.WriteLine("Clon: " + clienteClonado.empresa.nombre);

Console.WriteLine("Inicial: " + clienteOriginal.empresa.nombre);
```

Mostrar en la consola:

```
Inicial: Empresa Original

Clone: Empresa Original

Clone: Empresa Clonada

Inicial: Empresa Clonada
```

5. El tipo System.String y su alias string

Entre los tipos "integrados" en la gramática de C#, hay uno que es un poco particular: el tipo String (alias de System.String).

Este tipo representa una colección de caracteres Unicode, encapsulados por el tipo System.Char. String es de tipo Referencia (por lo tanto, asignado al heap) pero, por razones de comodidad, el uso del operador new para instanciarlo, no es el método utilizado de manera habitual. De hecho, basta con asignar una cadena durante la declaración de un objeto string para instanciarlo.

Instanciación de un objeto de tipo string

```
String s = "Viva el C#";
```

La cadena literal puede ser el resultado de una construcción.

```
    String hello = "Buenos días, es el "
            + DateTime.Today.DayOfYear
            + " º día del año";


    System.Console.WriteLine(hello);
```

Salida correspondiente por consola:

```
Buenos días, es el 156º día del año

Pulse una tecla para continuar...
```

Durante la declaración de la cadena literal, se utiliza por defecto el carácter \ (barra invertida) como secuencia de escape.

Por ejemplo, \r\n significa "retorno de carro y nueva línea", por tanto, incluye un salto de línea cuando se visualiza la cadena. Como consecuencia, si la cadena literal contiene ocurrencias reales del carácter \, entonces:

- aparecen dos veces o
- la declaración de la cadena se debe preceder del carácter @.

Ejemplo:

```
    string miArchivo1 = "C:\\temp\\miArchivo.txt";
    // o
    string miArchivo2 = @"C:\temp\miArchivo.txt";
```

@ también puede preceder a un nombre de variable homónima de una palabra clave del lenguaje; ejemplo con "continue": bool @continue = true;

Además de este modo de instanciación tan clásico, la clase String ofrece varios constructores con usos mucho más específicos.

¡Un tipo String puede contener hasta 2 GB de caracteres!

Sin ánimo de molestar a los desarrolladores de Java, el operador == se redefinió en la clase String para que pudiera comprobar valores y no las referencias.

```
    String s10 = "Hello";
    String s20 = "Hello";
    System.Diagnostics.Debug.Assert(s10 == s20);
```

Para los desarrolladores reticentes a este tipo de sintaxis existen los métodos Compare y CompareTo que se pueden utilizar para comparar el contenido de dos objetos String.

```
   String s10 = "Hello";
   String s20 = "Hello";

   System.Diagnostics.Debug.Assert( String.Compare(s10,s20)==0 );
   System.Diagnostics.Debug.Assert(  s10.CompareTo(s20) == 0  );
```

El valor de un objeto String es la cadena de caracteres que contiene. Esta cadena no se puede modificar. Incluso si la clase ofrece métodos que permiten realizar modificaciones del

contenido, hay que entender que la CLR va a recrear una nueva entidad del heap, porque el contenido original es inmutable.

El siguiente código muestra este comportamiento:

```
    class StringComprobador
    {


        public void Test()
        {
            string s1 = "Hola";
            string s2 = Modificar(s1, "Hello World");
            System.Diagnostics.Debug.WriteLine("S1=" + s1);
            System.Diagnostics.Debug.WriteLine("S2=" + s2);


        }


        string Modificar(string aModificar, string nuevoContenido)
        {
            aModificar = nuevoContenido;
            return aModificar;
        }


    }
```

La clase StringComprobador ofrece un método Test que instancia un objeto string s1 con el valor "Hola". Después lo busca para modificarlo en "Hello World".

Esta modificación se realiza en el método Modificar, que recibe una referencia a s1 como primer argumento y el nuevo valor como segundo argumento.

El tipo String es de tipo Referencia, y cualquier modificación realizada dentro del método de modificación debería replicarse en el objeto String original. De hecho, no ocurre así. El valor inicial permanece inmutable, la CLR crea una nueva cadena con una nueva referencia. Si la modificación se hubiera hecho directamente en el método Test, s1 habría contenido "Hello World" y el cambio de referencia hubiera sido "transparente". Como la modificación se realiza en un método, se genera una segunda referencia sobre s1 durante la llamada y esta es la "copia" que la CLR va a modificar, asignándole el nuevo valor. Para confirmar la manipulación, el método Modificar devuelve el resultado de la modificación que el método Test mantiene en el objeto s2. Conclusión: el original no se ha modificado como se puede comprobar en la ventana **Salida** de Visual Studio.

```
S1=Hola
```

```
S2=Hello World
```

Si una cadena de caracteres se debe componer dinámicamente, es mejor utilizar la clase StringBuilder que la clase String.

Ejemplo de código que se debe evitar:

```
    string haIntroducido = "Su nombre: ";
    System.Console.WriteLine("Indique su nombre");
    haIntroducido += System.Console.ReadLine();
    haIntroducido += "\r\nSu apellido: ";
    System.Console.WriteLine("Indique su apellido");
    haIntroducido += System.Console.ReadLine();
    haIntroducido += "\r\nSu edad: ";
    System.Console.WriteLine("Indique su edad");
    haIntroducido += System.Console.ReadLine();
    System.Console.WriteLine(haIntroducido);
```

y sustituir por:

```
    StringBuilder haIntroducido
        = new StringBuilder("Su nombre: ");
    System.Console.WriteLine("Indique su nombre");
    haIntroducido.Append(System.Console.ReadLine());
    haIntroducido.Append("\r\nSu apellido: ");
    System.Console.WriteLine("Indique su apellido");
    haIntroducido.Append(System.Console.ReadLine());
    haIntroducido.Append("\r\nSu edad: ");
    System.Console.WriteLine("Indique su edad");
    haIntroducido.Append(System.Console.ReadLine());
    string result = haIntroducido.ToString();
    System.Console.WriteLine(result);
```

Ejercicio corregido

1. Enunciado

Crear una aplicación de tipo Consola que servirá de soporte a las siguientes cuestiones:

Compruebe por programación que el tipo integrado int es el alias del tipo System.Int32.

Compruebe que int y System.Int32 son de tipo Valor.

Compruebe que string no es de tipo Valor.

Muestre en la consola el número de bytes utilizados para almacenar el valor de un tipo int.

2. Corrección

Seleccione el menú **Archivo** y después **Nuevo proyecto**. Indique la información solicitada.

El contenido del archivo fuente generado por el asistente es el siguiente:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Cap3
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Compruebe la igualdad de los tipos int y System.Int32 insertando siguiente el código en el Main:

```
    // Prueba sin instanciación
    System.Diagnostics.Debug.Assert(
        typeof(int) == typeof(System.Int32)
        );

    // Prueba con instanciación de los dos tipos
    int i = 3;
    System.Int32 j = 4;
    System.Diagnostics.Debug.Assert(
        i.GetType() == j.GetType()
        );

    // Prueba mixta
    System.Diagnostics.Debug.Assert(
        i.GetType() == typeof(System.Int32)
        );



    // Mostrar el tipo
    System.Diagnostics.Debug.WriteLine(i.GetType());
```

Compruebe que int y System.Int32 son de tipo Valor añadiendo el siguiente código a Main.

```
    int k = 5;
    System.Object obj1 = k;
    System.Diagnostics.Debug.Assert(obj1 is ValueType);
    System.Int32 l = 6;
    System.Object obj2 = k;
    System.Diagnostics.Debug.Assert(obj2 is ValueType);
```

Compruebe que string no es de tipo Valor añadiendo el siguiente código a Main.

```
    string s = "Hello";
    System.Object obj3 = s;
    System.Diagnostics.Debug.Assert(!(obj3 is ValueType));
```

Muestre el número de bytes utilizados para almacenar un int añadiendo el siguiente código a Main.

```
    System.Console.WriteLine(
        "Tamaño de un int: " + sizeof(int) + " bytes"
        );
```

El código completo es el siguiente:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Cap3
{
    class Program
    {
        static void Main(string[] args)
        {
            // Prueba sin instanciación
            System.Diagnostics.Debug.Assert(
                typeof(int) == typeof(System.Int32)
                );

            // Prueba con instanciación de los dos tipos
            int i = 3;
            System.Int32 j = 4;
            System.Diagnostics.Debug.Assert(
                i.GetType() == j.GetType()
```

```
                );

            // Prueba mixta
            System.Diagnostics.Debug.Assert(
                i.GetType() == typeof(System.Int32)
                );


            // Mostrar el tipo
            System.Diagnostics.Debug.WriteLine(i.GetType());


            int k = 5;
            System.Object obj1 = k;
            System.Diagnostics.Debug.Assert(obj1 is ValueType);
            System.Int32 l = 6;
            System.Object obj2 = k;
            System.Diagnostics.Debug.Assert(obj2 is ValueType);

            string s = "Hello";
            System.Object obj3 = s;
            System.Diagnostics.Debug.Assert(!(obj3 is ValueType));

            System.Console.WriteLine(
                "Tamaño de un int: " + sizeof(int) + " bytes"
                );
        }
    }
}
```

Genere la solución pulsando [F7].

Ejecute el programa pulsando [Ctrl][F5]. Si no hay ningún error de aserción en ninguna ventana, todas las pruebas Assert se han realizado correctamente. Sencillamente debe leer en la consola que un int (es decir, el alias de un System.Int32) ocupa 4 bytes, es decir 32 bits y, por tanto, tiene 2^32 posibilidades.

Introducción

Recordemos que una clase es un modelo que el sistema utiliza para instanciar en memoria el objeto correspondiente. Son los modelos que el desarrollador declara lo que comúnmente denomina archivos de código fuente (archivos de extensión .cs) de su proyecto. Es lo que hemos hecho con la clase Program.

Los espacios de nombres

El framework .NET ha organizado sus clases agrupándolas por su finalidad, en espacios de nombres (namespace), apareciendo estos en los ensamblados que se instalan en el GAC.

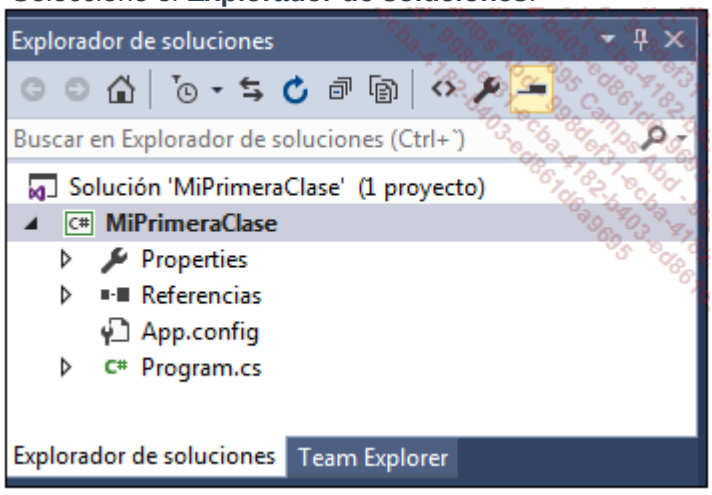
Como ellos, las clases que va a desarrollar deberán pertenecer a un espacio de nombre. Si omite su declaración, el sistema la creará por defecto. Es muy aconsejable hacerlo usted mismo, porque esto le va a permitir estructurar sus proyectos y controlar el "alcance" de los nombres, clases y métodos.

Elija un nombre de espacio de nombres que resuma el papel de la familia de clases que agrupa. Generalmente, el análisis UML le ayudará a su nomenclatura.

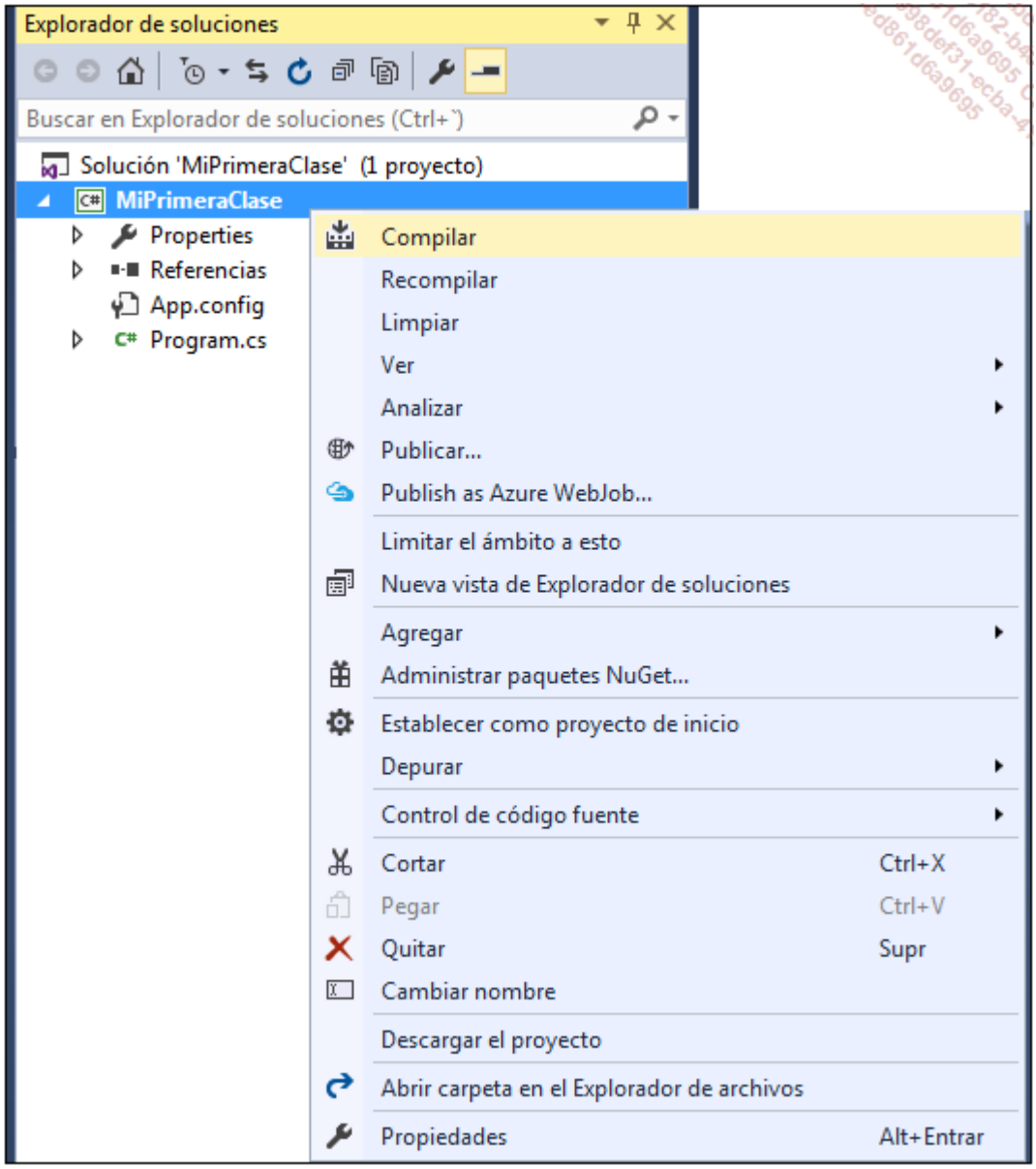
Este nombre debe empezar por una letra o por un guión bajo (_). A continuación puede contener letras, cifras y los guiones bajos. Evite utilizar caracteres acentuados y, si su definición contiene varias palabras, opte por la notación "tipo camello" (CamelCase). Por ejemplo, si escribe un espacio de nombres para una serie de herramientas que gestionan las facturas de compra, el nombre en formato "CamelCase" podría ser GestionFacturasCompras. Las primeras letras de las palabras relacionadas están en mayúscula.

Por defecto, Visual Studio utiliza el nombre del proyecto como nombre por defecto del espacio de nombres y también como nombre del ensamblado. Por supuesto, se puede modificar esta configuración.

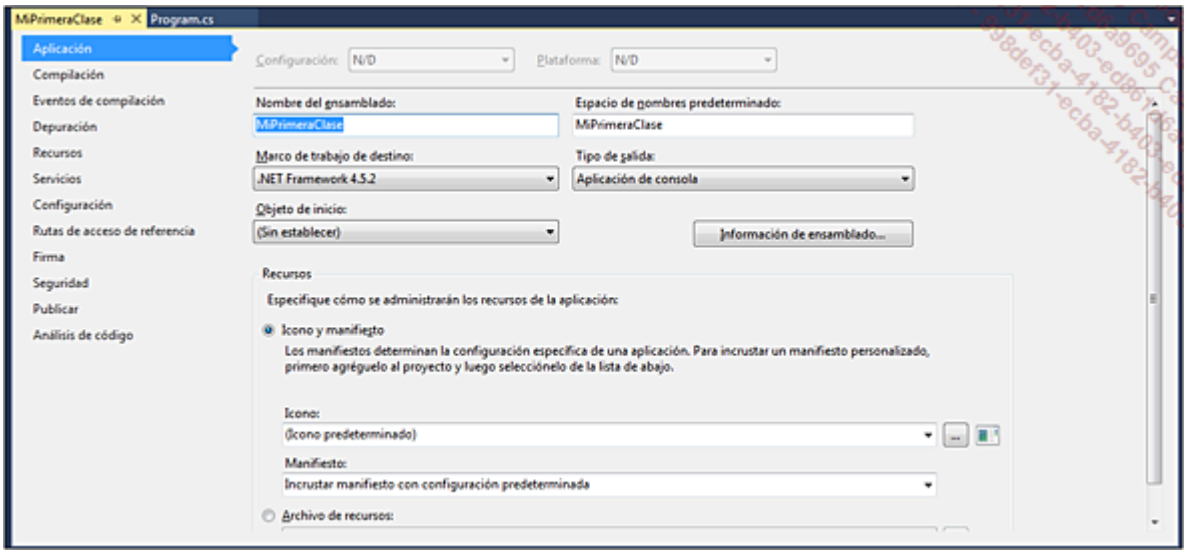
Seleccione el **Explorador de soluciones**.



Acceda a las propiedades del proyecto haciendo clic con el botón derecho del ratón en su entrada.



Modifique el nombre del ensamblado y/o del espacio de nombres por defecto:



La palabra clave namespace

Sintaxis de declaración:

```
namespace NombreNamespace
{
   // Cuerpo del namespace
}
```

La declaración del espacio de nombres se realiza mediante la palabra clave namespace, seguida del nombre que haya elegido darle. El código que se añade a este espacio de nombres se enmarca entre llaves.

Naturalmente, un mismo espacio de nombres se puede definir en varios archivos de código fuente.

```
namespace MiPrimerNamespace
{
   class MiPrimeraClase
   {
      //...
   }
}
```

La directiva using

En sus espacios de nombres, utilizará tipos que están en otros espacios de nombres, como los de .NET. Por ejemplo, podrá utilizar el tipo File y aprovechar sus servicios para la gestión de sus archivos. El tipo File se encuentra en el espacio de nombres System.IO, por lo que puede utilizarlo declarando su ruta completa:

```
System.IO.File.Copy("ArchivoOrigen.txt",
                    "ArchivoDest.txt", true);
```

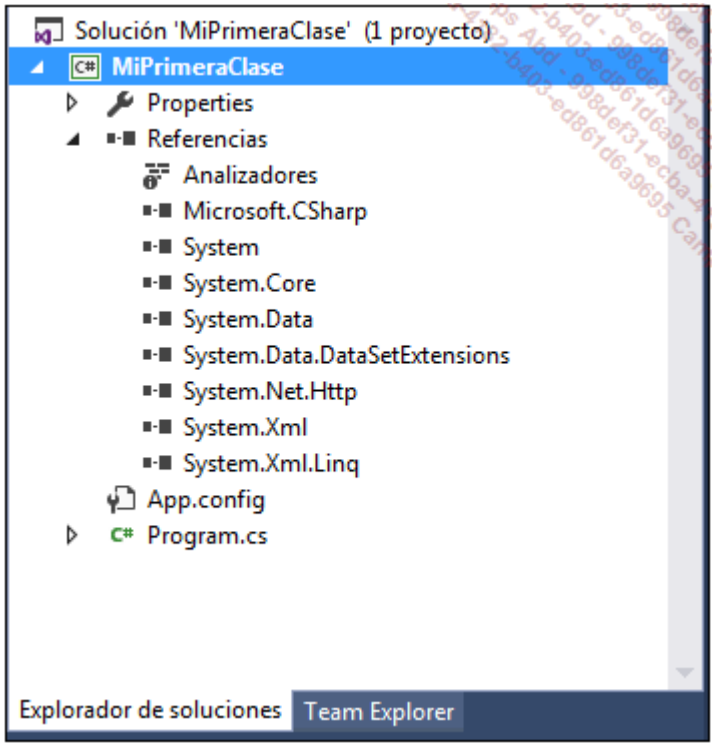
Si utiliza varias veces los servicios de la clase File y no desea repetir cada vez System.IO, puede definir en el encabezado del archivo fuente la directiva using System.IO;.

```
using System.IO;
namespace MiPrimerNameSpace
{
```

Al final, el encabezado clásico de un archivo de código C# contiene una serie de líneas que empiezan con la directiva using. Estas líneas dirigen al compilador a la lista de espacios de nombres que podrá consultar para identificar los tipos utilizados en este archivo.

Una directiva using solo es válida si el ensamblado (el assembly) que contiene el espacio de nombres asociado se referenció en el proyecto. Recordemos que un ensamblado puede contener tipos definidos en varios espacios de nombres.

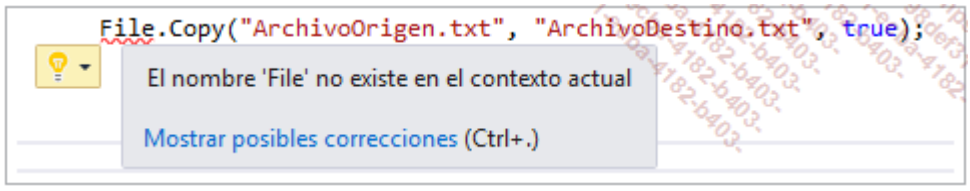
Los asistentes de creación de proyectos de Visual Studio hacen referencia a los ensamblados básicos, como puede ver en el **Explorador de soluciones**, abriendo la línea **Referencias**.



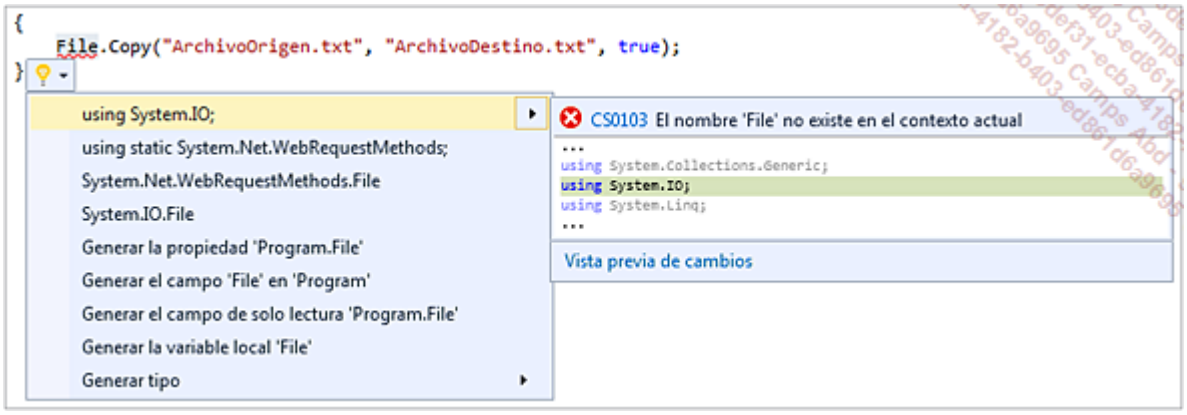
Observe que el asistente para la escritura del código le ofrece resolver los problemas de espacios de nombres. Por ejemplo, si el tipo File se utiliza sin declaración, entonces una línea ondulada roja lo subraya.

```
File.Copy("ArchivoOrigen.txt", "ArchivoDestino.txt", true);
```

Situando el cursor del ratón sobre el subrayado, revela el origen del error:

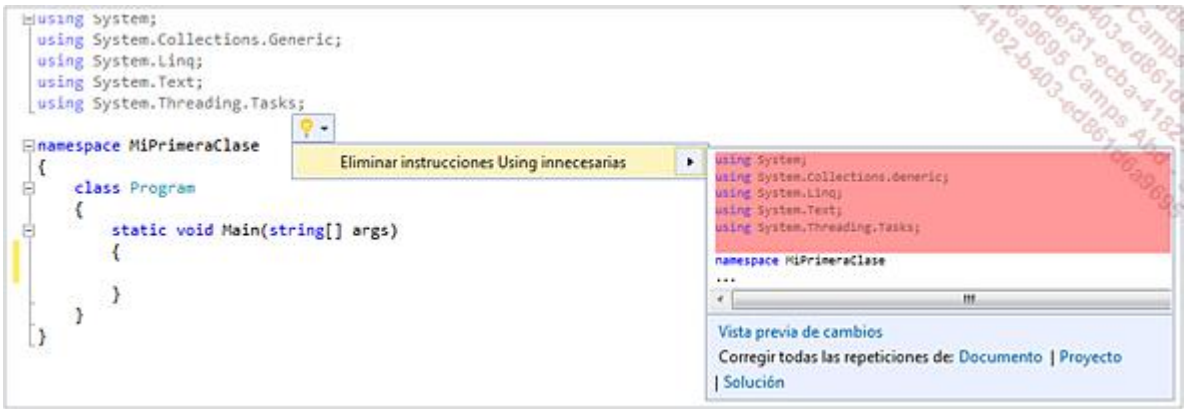


Un clic con el botón derecho del ratón en el subrayado, le ofrece varias soluciones para resolver el problema:



La primera solución consiste en añadir la directiva using System.IO; usted mismo en el encabezado del archivo. Las otras dos permiten prefijar el tipo File por su espacio de nombres completo. Observe que, en este ejemplo, existe un tipo File en dos espacios de nombres de .NET. Si se deben utilizar estos dos tipos en el mismo código fuente será necesario aclarar la ambigüedad usando en cada uno su espacio de nombres como prefijo.

Es muy frecuente tener encabezados que contengan directivas using que han dejado de ser útiles como consecuencia de modificaciones del código. Estas líneas inútiles se destacan. Visual Studio le puede ayudar a adecuar la lista a las necesidades reales. Para esto, pulse en una de las líneas de definición using y verá un icono en forma de bombilla que aparecerá en el margen. Pulse en esta bombilla para mostrar un menú contextual que contiene **Eliminar instrucciones Using innecesarias** a su derecha, mostrando una vista previa de la modificación.



Declaración de una clase

Una vez que se ha declarado el espacio de nombres, se puede definir la clase. Aunque esto generalmente es desaconsejable, es posible declarar varias clases del mismo nivel en un mismo archivo de código fuente. Una clase se declara con la palabra clave class seguida del nombre que ha elegido. Como para el namespace, este nombre debe empezar por una letra o por un guión bajo (_). A continuación puede contener letras, cifras y guiones bajos. Evite utilizar caracteres acentuados y opte por el formato "de tipo camello" (CamelCase). Por ejemplo, si escribe una clase que emula un lector digital, el nombre en formato "CamelCase" sería LectorDigital. Las primeras letras de las palabras relacionadas se escriben en mayúscula.

Aunque se explica en detalle más adelante, a continuación se realiza la declaración de una herencia. De hecho, si la clase extiende una clase de base y/o implementa una o varias interfaces, el nombre de la clase está seguido del signo ':' y a continuación la enumeración de sus componentes padres.

Los miembros de la clase (atributos, propiedades y métodos) se definen a continuación entre llaves.

Sintaxis de declaración:

```
visibilidad class NombreClase  [:[ClaseMadre], [Lista
interfaces de base]]
{
   // cuerpo de la clase
}
```

Ejemplo

```
namespace MiPrimerNameSpace
{
   class MiPrimeraClase: SuClaseMadre, Interfaz1, Interfaz2
   {
      // Cuerpo de la clase
   }
}
```

El atributo de visibilidad de una clase definido en un namespace puede ser de tipo:

- public: la clase se podrá utilizar por todos.
- internal: la clase solo se podrá utilizar por los componentes del assembly que la contengan.

Los demás atributos (private, protected y protected internal) solo tienen sentido para las clases anidadas. Una clase anidada es una clase definida en otra. Volveremos más tarde sobre el interés de este tipo de declaración.

La clase anidada precedida por el atributo private solo se podrá utilizar en su clase anfitriona.

La clase anidada precedida por el atributo protected solo se podrá utilizar por las clases heredadas de su clase anfitriona.

La clase anidada precedida por el atributo protected internal solo se podrá utilizar por las clases heredadas de su clase anfitriona en el ensamblado.

Las clases anidadas (nested clases) se presentan más adelante.

1. Accesibilidad de los miembros

El nivel de accesibilidad de los miembros de una clase se define por un "modificador de acceso", que precede a la declaración de cada miembro.

Este modificador de acceso puede ser:

- public, para un acceso sin restricción.
- protected, para un acceso limitado a la clase y a sus clases heredadas.
- internal, para un acceso limitado al ensamblado actual.
- protected internal, para un acceso limitado al ensamblado actual y a las clases que heredan de la clase.
- private, para un acceso limitado al tipo de la clase.

Si no indica la visibilidad de un miembro de la clase, será de tipo private.

2. Atributos

Los atributos (también llamados variables) de la clase se deben declarar en el interior de las llaves de la clase.

C#, como Java, es un lenguaje "full object" y es imposible encontrar una variable definida fuera de una declaración de tipo. Es una de las diferencias significativas con C++ que, teniendo que ser compatible con C, ha tenido que seguir dando soporte a una zona de definición "global".

Respetar la encapsulación de la programación orientada a objetos debe conducir al desarrollador a limitar el nivel de accesibilidad de los atributos de sus clases.

Lo mejor es configurar todos los atributos de sus clases con un acceso de tipo privado.

Un atributo se define por su visibilidad, tipo (entero, cadena, referencia a otro objeto...) y nombre. Eventualmente, el atributo se puede inicializar directamente en su definición (novedad bastante amigable respecto a C++). El carácter ';' termina la definición.

El nombre del atributo sigue las mismas restricciones que las del nombre de la clase o del namespace. La notación CamelCase siempre está de moda, pero la primera letra del atributo generalmente está en minúscula. Veremos un poco más adelante por qué.

Sintaxis de declaración:

```
visibilidad tipo miAtributo [=valor o referencia];
```

Ejemplo:

```
namespace MiPrimerNameSpace
{
   class MiPrimeraClase
   {
      private int porcentajeDescuento = 20;
      private string nombreProducto;
   }
}
```

Generalmente, un atributo se modificará durante todo el ciclo de vida de la clase continente. Como vamos a descubrir en la siguiente sección, es posible "congelar" el contenido bien durante su definición, con la palabra clave const o durante la "construcción " de la clase, con la palabra clave readonly.

a. Atributos constantes

Es muy frecuente que un programa necesite datos constantes, introducidos en el momento de la declaración de la clase. Por ejemplo, el nombre que le da a su programa se podrá utilizar en varios lugares, y esto en modo solo lectura. No sería apropiado tener que "duplicar" este nombre cada vez que se utiliza, porque el día que se tuviera que cambiar sería necesario repetir la modificación en todos los sitios.

C y C++ ofrecen una sintaxis basada en el uso de la palabra clave #define.

Ejemplo:

```
#define PORCENTAJE_DESCUENTO 20
```

El compilador sustituye todas las ocurrencias de PORCENTAJE_DESCUENTO por 20.

El problema de esta solución es que PORCENTAJE_DESCUENTO está débilmente tipado. No hay duda de que se trata de un tipo numérico pero, ¿es un int, un uint, un long, un float o incluso un double?

Con C#, el problema queda resuelto porque se debe definir el tipo del dato constante, utilizando como prefijo la palabra clave const para impedir cualquier modificación posterior.

Sintaxis de declaración:

```
visibilidad const tipo miAtributo =valor;
```

Ejemplo:

```
namespace MiPrimerNameSpace
{
   class MiPrimeraClase
   {
      private const string nombreProducto = "Aplicación C#";
   }
}
```

Cualquier intento de modificación de un atributo constante provoca un error de compilación.

b. Atributos en modo solo lectura

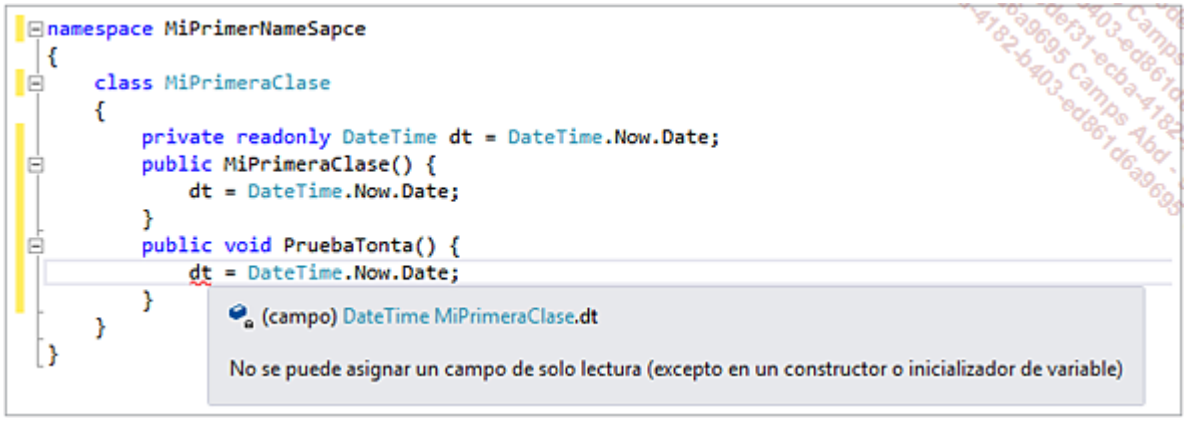
El problema del atributo constante es que su valor se debe conocer durante su declaración en la clase. Imagine que queremos memorizar de manera inalterable la dirección IP de la máquina, que se lee durante la instanciación de un objeto y no al cargar la aplicación; la sintaxis basada en la palabra clave const no sería conveniente. En este caso habría que utilizar el atributo readonly.

La palabra clave readonly indica que el atributo se puede inicializar tanto durante su declaración con const como en un constructor de la clase que lo contiene. Hay explicaciones completas de los constructores y su sintaxis un poco más adelante, pero de momento es suficiente con saber que los constructores son los métodos de clase que se llaman desde la instanciación del objeto en memoria. El papel de los constructores de clase es esencialmente el de definir los valores de inicio de los atributos. Para solucionar el problema anterior, el constructor puede preguntar a .NET y pedirle la dirección IP asignada a la máquina para memorizarla después de manera definitiva en un atributo que utiliza el prefijo readonly.

Sintaxis de declaración:

```
visibilidad readonly tipo miAtributo [=valor];
```

Ejemplo:



Este fragmento de código muestra las dos formas de inicialización posibles de un atributo readonly: en el inicializador de variable o en el constructor de la clase (se presenta más adelante).

Observe que el método PruebaTonta provoca un error de compilación porque intenta modificar el contenido del atributo readonly fuera de los dos casos permitidos.

3. Propiedades

Si los atributos de los objetos son privados, ¿cómo podrían los usuarios leerlos y modificarlos? Hay que rodear los atributos de una "capa" protectora que comprobará que los valores que los usuarios desean darle están permitidos. Para esto, hay dos posibles codificaciones.

Solución de los descriptores de acceso/modificadores de tipo en Java pero que también funciona en C#

Se trata de ofrecer para cada atributo que se desee proteger una pareja de métodos que permitan leer y escribir. En el mundo Java, a esto se le llama descriptores de acceso, con una especificación "getter" para la lectura y "setter" para la escritura.

Ejemplo:

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {
        private int porcentajeDescuento;
        public bool SetPorcentajeDescuento(int value)
        {
           bool bRet = false;
           //Prueba si el argumento que se pasa
           //está en los límites correctos
           if (value >= 0 && value < 100)
           {
              porcentajeDescuento = value;
              bRet = true;
           }
           return bRet;
       }

       public int GetPorcentajeDescuento()
       {
           return porcentajeDescuento;
      }
   }
}
```

Del lado del usuario de la clase, la redacción del código es un poco larga, pero reconozcamos que los nombres de los métodos tienen la virtud de ser muy claros.

Solución C#

C# ofrece las propiedades (properties) que dan, del lado del usuario de la clase, la apariencia de acceder directamente a los atributos y, del lado de la propia clase en sí misma, el respecto total a la regla de la encapsulación. A continuación vamos a ver, usando un ejemplo, cómo funcionan las propiedades.

A continuación se muestra el código anterior transformado a la "manera propiedades":

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {
       private int porcentajeDescuento;
       public int PorcentajeDescuento
       {
           get
           {
               return porcentajeDescuento;
           }
           set
           {   //Prueba si el argumento que se pasa
               //está en los límites permitidos
               if (value >= 0 && value < 100)
                   porcentajeDescuento = value;
           }
        }
    }
}
```

En este código vemos la definición del atributo de tipo private, seguido por la definición de la propiedad. Al inicio, la definición de la propiedad se parece a la del atributo. Encontramos el atributo de visibilidad, que se pasa generalmente a public, el tipo, que debe corresponder con el del atributo que encapsula y el nombre. Por convención, el nombre de la propiedad corresponde con el del atributo, con el primer carácter en mayúscula. Esta correspondencia entre los nombres es una sencilla comodidad; si desea tener nombres totalmente diferentes resultará menos legible aunque sería perfectamente posible. A continuación vienen dos bloques, get y set, correspondientes a la lectura y la escritura del atributo privado. Es posible trasladar al bloque set las comprobaciones y validaciones de uso para el atributo asociado.

La sintaxis puede parecer un poco pesada. Veamos a continuación cómo Visual Studio nos ayuda en su escritura.

A continuación se muestra la clase, vacía:

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {


    }
}
```

Queremos añadir en esta clase un atributo protegido de tipo entero, llamado porcentajeDescuento.

Para arrancar el asistente de escritura del atributo y su propiedad, hay que escribir propfull por el teclado.

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {
          propfull
    }
}
```

... después de pulsar en la tecla de tabulación, se obtiene el siguiente resultado:

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {
        private int myVar;
        public int MyProperty
        {
          get { return myVar;}
          set { myVar = value;}
        }
    }
}
```

Si el tipo int no se corresponde con el tipo que desea definir, modifíquelo y pulse de nuevo la tecla tabulación para obtener el nombre de la propiedad. Rellene el nombre de esta propiedad utilizando o no las convenciones de uso.

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {
        private int porcentajeDescuento;

        public int PorcentajeDescuento
        {
            get { return porcentajeDescuento; }
            private set { porcentajeDescuento = value; }
        }
    }
}
```

A continuación, es "suficiente" con insertar el código de comprobación en el bloque set para añadir las condiciones que quiera.

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {
        private int porcentajeDescuento;
        public int PorcentajeDescuento
        {
            get
            {
                return porcentajeDescuento;
            }
            set
            {   //Comprueba si el argumento que se pasa
                //está en los límites permitidos
                if (value >= 0 && value < 100)
                    porcentajeDescuento = value;
            }
        }
    }
}
```

La redacción del código se simplifica mucho.

Desde el exterior, el uso de la propiedad parece un acceso directo al atributo. La única ventaja del enfoque de tipo Java: el retorno binario del setter informa al usuario de la clase de si el valor que se pasa es o no válido. Veremos que será posible utilizar el mecanismo de las excepciones para permitir a la clase utilizada reportar un error al código que llama, incluso durante la escritura de propiedades.

Si el papel de una propiedad concreta consiste solo en leer y escribir el contenido de un atributo, sin realizar controles particulares, será posible utilizar una sintaxis más ligera en la que el atributo no se define explícitamente. En este caso, hablamos de propiedad "automática".

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {
        public int PorcentajeDescuento { get; set; }
    }
}
```

De nuevo, Visual Studio nos puede ayudar en esta creación. Para ello, esta vez es suficiente con escribir Prop...

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {
        prop
    }
}
```

...pulsar la tecla [Tab], ...

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {
        public int MyProperty { get; set; }
    }
}
```

... modificar eventualmente el tipo e indicar el nombre de la propiedad.

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {
        public int PorcentajeDescuento { get; set; }
    }
}
```

Esta sintaxis alcanzará sus límites tan pronto como quiera añadir operaciones.

A partir de la versión 6 de C# es posible inicializar el contenido de una propiedad durante su declaración en la clase.

```
    class MiPrimeraClase
    {
         public int PorcentajeDescuento { get; set; } = 25;
    }
```

Propiedades en modo solo lectura

El párrafo anterior ha presentado dos métodos que permiten definir un atributo en modo solo lectura.

Puede ser oportuno hacer que determinadas propiedades de su clase estén en modo solo lectura para sus usuarios, conservando su capacidad de ser modificables en la clase. Se trata de un comportamiento más sofisticado que el de const o readonly, porque el atributo sigue siendo modificable.

Con el método "descriptor de acceso Java", basta con no definir el método Setxxx. Con las propiedades C#, basta con preceder set por un private, como se muestra a continuación:

```
namespace MiPrimerNameSpace
{
    class MiPrimeraClase
    {
        private int porcentajeDescuento;
        public int PorcentajeDescuento
        {
            get
            {
                return porcentajeDescuento;
            }
            private set
            {
                porcentajeDescuento = value;
            }
        }
    }
}
```

En este fragmento de código, el usuario de la clase solo podrá leer la propiedad PorcentajeDescuento. Cualquier intento de escritura de la propiedad generará un error de compilación.

Los atributos de tipo "static"

Los atributos y las propiedades generalmente están asociados a la instancia de la clase que los ha definido. Por ejemplo, si ha creado una clase Cliente, es probable que contenga un atributo que lo identifica, por lo que el contenido será diferente para cada instancia Cliente. Imaginemos un sistema de numeración sencillo: el primer cliente tendrá el identificador 1, el segundo el identificador 2, y así consecutivamente.

Para generar esta progresión, la mayor parte del tiempo necesitará un contador de instancias creadas (la mayor parte del tiempo, porque si utiliza los servicios de una base de datos podrá, en efecto, soportar esta numeración única). O bien declara este contador en un objeto de gestión de las instancias de clase Cliente, que se creará antes que cualquier instancia de Cliente y después se llamará con la creación de cada nuevo cliente, o bien declara este contador como miembro de tipo static del tipo Cliente.

En este último caso, y este es el caso que nos interesa, el contador se adjunta al tipo Cliente y no a una instancia de Cliente; por tanto, se comparte por todas las instancias.

Sabemos que el constructor de la clase es un método que se llama para inicializar los atributos de la clase. Cada vez que se crea una nueva instancia Cliente, este constructor podrá leer el valor del contador de tipo static, incrementarlo y copiar su contenido en su atributo identificador.

A continuación se muestra el código asociado a este tipo de uso:

```
class Cliente
{
    private static int contadorClientes = 1;
    private int identificadorCliente;
    public int IdentificadorCliente
    {
        get { return this.identificadorCliente; }
        private set { this.identificadorCliente = value; }
    }
    public Cliente()
    {
        this.identificadorCliente = Cliente.contadorClientes;
        Cliente.contadorClientes++;
    }
}
```

Observe que contadorClientes se define a 1 en la clase Cliente. Esta asignación se realiza solo durante la primera instanciación de la clase Cliente. El capítulo siguiente ofrece otro método que permite inicializar los atributos de tipo static para que puedan recibir valores definidos durante la ejecución y no durante la compilación.

Observe también que para acceder a un dato de tipo static, necesariamente hay que volver a llamar a la clase que lo contiene, como aquí con Cliente.contadorClientes.

4. Constructor

a. Etapas en la construcción de un objeto

Cuando un desarrollador desea crear una instancia de su clase Cliente, utiliza la palabra clave new seguida del tipo Cliente.

```
namespace MiPrimerNameSpace
{
    class Program
    {
        static void Main(string[] args)
        {
            // Inicio de las operaciones
            Cliente miCliente = new Cliente();
            // ...resto de operaciones
        }
    }
}
```

¿Qué sucede concretamente durante esta ejecución?

- La CLR pide al sistema operativo una "sección" de memoria del tamaño de un objeto Cliente.
- Ejecuta el constructor de cada atributo definido en la clase, lo que provoca una inicialización por defecto (por ejemplo, los valores numéricos a 0).
- Busca si se ha definido un constructor para esta clase Cliente y, si es el caso, lo llama.
- Devuelve al programa que llama una referencia al objeto que se acaba de asignar.

La segunda etapa es una mejora de C++, que se contenta con indicar el bloque "bruto" asignado por el sistema operativo. Si el desarrollador obliga a inicializar los atributos de su clase, entonces se llevará sorpresas durante la ejecución. Con C#, esta fase de inicialización sistemática hace prácticamente opcional la implementación de un constructor, salvo si se necesita un cálculo específico, como la asignación de un identificador de cliente.

Sintaxis de declaración de un constructor:

```
visibilidad NombreDeLaClase([argumentos])
{
     // Implementación
}
```

La sintaxis de un constructor empieza definiendo su visibilidad, que puede ser:

- public, para autorizar a todo el mundo a crear este tipo de objetos.
- protected, para un acceso limitado a las clases heredadas.
- internal, para un uso limitado desde los objetos del ensamblado actual.
- protected internal, para un acceso limitado desde los objetos del ensamblado actual y para un acceso limitado a las clases heredadas.
- private, para prohibir la instanciación de este tipo. Más adelante veremos con más detalle el interés de una configuración como esta.

Si no indica la visibilidad del constructor, será de tipo private.

Al contrario de lo que sucede con un método "clásico", un constructor no devuelve nada. Por tanto, tenemos directamente después de la definición de su visibilidad el nombre de la clase seguido de un paréntesis abierto y otro paréntesis cerrado. La operación del constructor se codifica a continuación entre llaves.

```
class Cliente
{
    public Cliente ()
    {
        //...
    }
}
```

b. Sobrecarga de constructores

Un constructor se puede "sobrecargar", es decir, tener diferentes versiones con diferentes configuraciones.

Si un constructor debe recibir argumentos, se declararán entre los paréntesis.

```
class Cliente
{
    public Cliente ()
    {
        //...
    }
    public Cliente (string nombre, bool activo)
    {
        //...
    }
}
```

c. Constructores con valores de argumentos por defecto

Los valores de sus argumentos pueden ser predefinidos, es decir, que si el elemento que invoca al constructor no los define se utilizarán los valores por defecto previstos por el diseñador.

En el siguiente ejemplo el argumento activo se convierte en opcional durante la llamada al constructor de la clase Cliente. En caso de omisión, se considerará como con valor true.

```
class Cliente
{
    public Cliente ()
    {
    }
    public string Nombre { get; set; }
    public bool Activo { get; set; }
    public Cliente (string nombre, bool activo=true)
    {
        Nombre = nombre;
        Activo = activo;
    }
}
```

d. Encadenamiento de constructores

Un constructor puede llamar a otro constructor de la misma clase (y/o un constructor de una clase de base). De esta manera, es sencillo formar una serie de operaciones evitando la redundancia de código.

El siguiente fragmento de código muestra este mecanismo. Un primer constructor sin argumento permite generar el identificador único de un cliente. Un segundo constructor permite recuperar su nombre y su estado de cliente activo. Problema: es necesario que la llamada del segundo constructor también genere un identificador.

Gracias a la secuencia :this() después de la definición es posible encadenar el constructor con argumentos sobre el constructor sin argumento.

```
class Cliente
{
    private static int contadorClientes = 0;
    private int identificadorCliente;
    public int IdentificadorCliente
    {
        get { return identificadorCliente; }
        private set { identificadorCliente = value; }
    }
    public Cliente()
    {
        this.identificadorCliente = Cliente.contadorClientes;
        Cliente.contadorClientes ++;
    }
    public string Nombre { get; set; }
    public bool Activo { get; set; }
    public Cliente (string nombre, bool activo=true)
       : this()
    {
        Nombre = nombre;
        Activo = activo;
    }
}
```

Este principio se puede extender a cualquier otro encadenamiento que acepte argumentos.

e. Los constructores de tipo static

Un constructor puede ser de tipo static. Esta forma se utiliza para inicializar los datos estáticos de la clase, con valores que se pueden evaluar durante la ejecución (a diferencia de con los valores fijados durante la compilación).

En el caso de un constructor de tipo static, no hay atributo de visibilidad. El constructor de tipo static se llama una única vez antes de la instanciación del primer objeto.

El siguiente fragmento de código muestra la inicialización del contador de clientes a 1.000 a través de un constructor estático.

```
class Cliente
{
    private static int contadorClientes  = 0;
    static Cliente()
    {
        contadorClientes  = 1000;
    }

    private int identificadorCliente;
    public int IdentificadorCliente
    {
        get { return identificadorCliente; }
        private set { identificadorCliente = value; }
    }
    public Cliente()
    {
        this.identificadorCliente = Cliente.contadorClientes;
        Cliente.contadorClientes++;
    }
    public string Nombre { get; set; }
    public bool Activo { get; set; }
    public Cliente (string nombre, bool activo=true)
       : this()
    {
        Nombre = nombre;
        Activo = activo;
    }
}
```

f. Los constructores de tipo private

¿Para qué puede servir una clase cuyo constructor es privado?

... Para prohibir su instanciación directa con un new fuera de la clase.

Pero, ¿con qué objetivo?

... Porque una clase como esta quiere gestionar ella misma su instanciación en la aplicación para, por ejemplo, figurar una única vez en memoria. Este método es la implementación de un design pattern, más conocido con el nombre de singleton.

El acceso de esta clase a una instanciación tan particular se realiza por medio de un método de tipo static, generalmente llamado GetInstance y que se encarga de instanciar el objeto durante su primera llamada.

Ejemplo de clase de tipo singleton:

```
// La clase "singleton"
  public class ClaseParaInstanciaUnica
  {
    // El constructor es privado para prohibir
    // un new ClaseParaInstanciaUnica() externo
    private ClaseParaInstanciaUnica()
    {
    }

    // Miembro privado que contiene una referencia a un
    // objeto de tipo... ClaseParaInstanciaUnica
    private static ClaseParaInstanciaUnica
                        instanciaUnica = null;

    // Método de tipo public static encargado de
    // crear "una única vez"
    // una instancia de ClaseParaInstanciaUnica
    public static ClaseParaInstanciaUnica GetInstance()
    {
      if (ClaseParaInstanciaUnica.instanciaUnica == null)
      {
        ClaseParaInstanciaUnica.instanciaUnica
              = new ClaseParaInstanciaUnica();
```

```
      }
      return instanciaUnica;
    }

    // Propiedad instanciada de tipo string
    public string MiCadena { get; set; }

    // Método que actúa sobre la propiedad instanciada
    public void AccionSobreUnaInstanciaUnica(string aAñadir)
    {
      this.MiCadena += aAñadir;
    }

    // Método que muestra el contenido de la propiedad instanciada
    public override string ToString()
    {
      return this.MiCadena;
    }
  }
```

Ejemplo de uso de esta clase singleton:

```
class Program
{
  static void Main(string[] args)
  {
    // La primera llamada a ClaseParaInstanciaUnica.GetInstance()
    // va a crear automáticamente la instancia
    ClaseParaInstanciaUnica claseParaInstanciaUnica
      = ClaseParaInstanciaUnica.GetInstance();
    claseParaInstanciaUnica.AccionSobreUnaInstanciaUnica("Hello ");

    // Las siguientes llamadas solo devolverán la referencia
    // única del objeto ClaseParaInstanciaUnica
    ClaseParaInstanciaUnica claseParaInstanciaUnica2
```

```
      = ClaseParaInstanciaUnica.GetInstance();
    claseParaInstanciaUnica2.AccionSobreUnaInstanciaUnica("World");


    // GetInstance()también se puede utilizar directamente
    // en una línea de código
    Console.WriteLine(
      ClaseParaInstanciaUnica.GetInstance().ToString()
      );
  }
}
```

Salida correspondiente por la consola:

```
Hello World
Pulse una tecla para continuar...
```

g. Los inicializadores de objetos

Esta vez, del lado del usuario de su clase, el inicializador de objeto es una sintaxis particular de creación de objetos seguida por una inicialización de sus propiedades. Este método ofrece una gran flexibilidad de uso, que permite disminuir de manera significativa el número de constructores sobrecargados de sus clases.

Sintaxis del inicializador de objeto:

```
Tipo nombre = new Tipo()
{
   Propiedad01 = valor01,
   Propiedad02 = valor02,
   ...
};
```


Ejemplo de creación de un objeto Cliente con inicialización de sus dos propiedades:

```
    Cliente miCliente = new Cliente()
    {
        Nombre = "Ángel",
        Activo = true
    };
```

Este código utiliza el constructor sin argumento, lo que va a generar el identificador único e inicializar posteriormente las propiedades. Cuestiona el desarrollo del constructor sobrecargado, presentado anteriormente.

Problema de esta solución: el contenido entre llaves se considera como un grupo, que la herramienta de puesta a punto del programa no puede "trazar" línea a línea. Por tanto, si una inicialización presenta algún problema, será un poco más difícil identificarlo.

5. Destructor

El destructor es el método de la clase que llama a la CLR justo antes de la des-asignación del objeto. Hablamos de "finalización".

En C++, el desarrollador destruye él mismo los objetos asignados a la cola después de su uso y, por tanto, controla en qué momento se invoca al destructor. Este principio funciona perfectamente, con la condición de no olvidar llamarlo.

En C#, como en Java, es el garbage collector (recolector de basura) el que gestiona este trabajo y, por tanto, el desarrollador no es responsable de esta tarea. Por el contrario, pierde el control del momento de la destrucción. Por tanto, si se debe ejecutar un código de "limpieza" rápida y sistemáticamente al final del uso de una instancia, se aconseja escribirlo en un método público llamado habitualmente Dispose y documentar su uso.

Si el usuario de la clase piensa llamar a este método Dispose al final del uso, el código de limpieza se ejecuta en el momento correcto. Además, será posible acelerar al reprocesamiento, informando al garbage collector de que ya no necesita "finalizar" el objeto y, por tanto, es inútil llamar al código de su destructor.

Si el usuario olvida utilizar el método Dispose y el objeto se detecta como no utilizado, entonces la CLR llamará al código del destructor antes de la desasignación del objeto.

El destructor y método Dispose son opcionales. Se deben implementar si el objeto ha utilizado recursos no gestionados o si se realizan actualizaciones de los atributos estáticos.

Un destructor es único en una clase; no tiene atributo de visibilidad, no recibe ningún argumento y no devuelve nada.

Sintaxis de un destructor:

```
~NombreDeLaClase()
{
   //...
};
```

Para ilustrar esta sección, a continuación se muestra el código de una clase que implementa el método Dispose y su destructor.

```
    class MiClase:IDisposable
    {   // MiClase implementa la interfaz IDisposable
        // En consecuencia, ofrece un método Dispose

        // Propiedad contador de instancias de MiClase
        public static int ContadorClientes { get; private set; }

        static MiClase()
        {   // El constructor static inicializa el contador a 0
            MiClase.ContadorClientes = 0;
        }

        public MiClase()
        {   // El constructor de instancia incrementa el contador
            MiClase.ContadorClientes++;
        }

        private bool Disposed = false;
        public void Dispose()
        {   // El método Dispose se ’debería’ llamar
            // cuando el programa ya no necesite
            // la instancia.
            if (!Disposed)
            {   // Dispose no ha sido llamado
                // Decrementa el contador
                MiClase.ContadorClientes--;
                // Declara el objeto como "finalizado"
```

```
                GC.SuppressFinalize(this);
                // registra el paso al método
                Disposed = true;
            }
        }

        ~MiClase()
        {   // Si este método se llama, el método
            // Dispose no se ha llamado
            Dispose();
        }

        public int MiPropiedad { get; set; }
        public void MiMetodo() { }

        //...
    }
```

6. Otro uso de using

La palabra clave using, ya conocida como directiva de definición de namespace, también se utiliza para reducir el código de uso de un objeto a "limpiar" al final de su uso.

En primer lugar, a continuación se muestra la asignación clásica de un objeto, seguida de su limpieza

```
    // Sintaxis ’clásica’
    MiClase mc = new MiClase();

    // Utilización de la clase
    //...
    // Fin de uso, pensamos en llamar a Dispose
    mc.Dispose();

    // mc se "limpia" pero todavía no se desasigna
    // será el garbage collector el que lo hará más tarde
```

A continuación se muestra la sintaxis con using, que evita la llamada a Dispose

```
    // Sintaxis con using
    using (MiClase mc = new MiClase())
    {
        // Utilización de la clase
        //...

    }   // inútil llamar a Dispose, la llave cerrada de
        // using lo hace por nosotros.
```

El uso de using solo es posible si el objeto asignado implementa la interfaz IDisposable. Si este no fuera el caso, se producirá un error de compilación.

Todavía no hemos estudiado las interfaces, pero por el momento sepa que esto vuelve a mostrar la certeza de que un objeto implementa el o los métodos descritos en la interfaz. En nuestro caso, IDisposable obliga a la instancia a "exponer" un método Dispose.

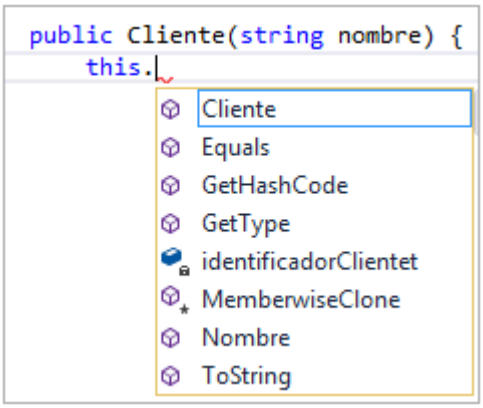
7. La palabra clave this

Ya hemos utilizado this dos veces y ha llegado el momento de concretar un poco más su sentido.

C# (como Java y C++) ofrece la palabra clave this para acceder a la instancia actual.

Atención, en C#, como en Java, this se utiliza con un punto para acceder a los miembros (this.identificador = 523). En C++ se utiliza con una flecha (this->m_identificador = 523;).

Visual Studio nos ayuda en la redacción de nuestro código; el hecho sencillo de poner this. en un método de la clase invoca al asistente IntelliSense que ofrece todos los miembros relacionados con la instancia:



Normalmente el uso de this es opcional y algunas veces puede entorpecer la lectura del código.

```
    public Cliente(string nombre)
    {
        this.Nombre = nombre;
        // equivalente a:
        Nombre = nombre;
    }
```

A pesar de todo, si una clase contiene miembros de tipo static, por tanto miembros adjuntos al tipo de la clase y no a su instancia, el uso de this puede permitir una mejor comprensión del código.

En el siguiente fragmento se ve claramente que el miembro adjunto a la instancia de la clase Cliente es identificadorCliente y el miembro adjunto al tipo Cliente es contadorClientes.

```
    public Cliente()
    {
        this.identificadorCliente = Cliente.contadorClientes;
        Cliente.contadorClientes++;
```

```
    }
```

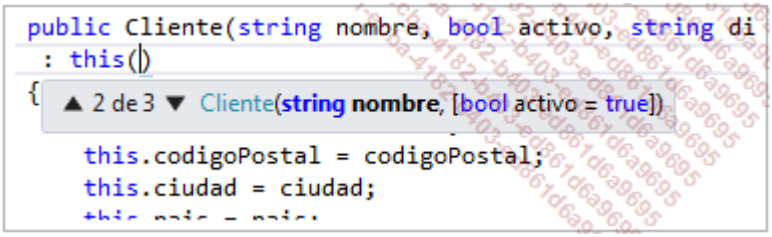
El uso de this. puede eliminar ambigüedades cuando hay nombres idénticos de variables.

En el siguiente fragmento se han elegido nombres comunes para los atributos y los argumentos asociados al constructor. Es una práctica muy común, posible gracias al uso de this.

```
    private string direccion;
    private string codigoPostal;
    private string ciudad;
    private string pais;

    public Cliente(string nombre, bool activo, string direccion,
                   string codigoPostal, string ciudad, string pais)
    {
        this.direccion = direccion;
        this.codigoPostal = codigoPostal;
        this.ciudad = ciudad;
        this.pais = pais;
    }
```

Hemos visto que this también permite encadenar constructores y evitar la duplicación de código.

En la siguiente imagen se comprueba que durante la introducción del código, el asistente ofrece todos los constructores disponibles después de introducir :this(.



El siguiente fragmento de código muestra el uso de this para realizar el encadenamiento entre tres constructores y, de esta manera, evitar código redundante.

```
namespace DemoEncadenamientoCtor
{
  class Program
  {
```

```
  static void Main(string[] args)
  {
    Cliente c = new Cliente(
            "Ángel", true,
            "Calle Islas Pitiusas", "12345",
            "Las Rozas", "Madrid");
    //...
  }
}

class Cliente
{
  private string Nombre;
  private bool Activo;
  private string direccion;
  private string codigoPostal;
  private string ciudad;
  private string pais;
  static int contadorClientes = 1000;
  private int identificadorCliente;

  public Cliente(string nombre, bool activo, string direccion,
                string codigoPostal, string ciudad, string pais)
   : this(nombre, activo)
  {
    this.direccion = direccion;
    this.codigoPostal = codigoPostal;
    this.ciudad = ciudad;
    this.pais = pais;
  }


  public Cliente(string nombre, bool activo = true)
```

```
     : this()
    {
      Nombre = nombre;
      Activo = activo;
    }

    public Cliente()
    {
      this.identificadorCliente = Cliente.contadorClientes;
      Cliente.contadorClientes++;
    }
  }
}
```

Durante la ejecución, el encadenamiento es tal que es el código de Cliente() el que se ejecuta en primer lugar. Después se ejecuta el de Cliente(string nombre, bool activo = true) y, por último, Cliente(string nombre, bool activo, string direccion, string codigoPostal, string ciudad, string pais).

8. Métodos

Ya se han implementado algunos y podemos adivinar que los métodos contienen las operaciones de una clase. Son las secciones de código que se ejecutan por el propio objeto, internamente, o desde otros objetos que tienen los permisos necesarios. Todos los métodos, incluido Main - punto de entrada de la aplicación - se encapsulan en las definiciones de clase.

a. Declaración

Sintaxis de un método:

```
[Atributo visibilidad][Modificador]<tipo de retorno><Nombre>
([tipo argumento], [tipo argumento 2],...)
{
<Código>;
```

```
<Código>;
}
```

Los atributos de visibilidad

Como el resto de miembros de la clase, los métodos generalmente se preceden de un atributo de visibilidad:

- public, para autorizar a todo el mundo a utilizar el método.
- protected, para un acceso limitado a las clases heredadas de la clase.
- internal, para un uso limitado a los objetos del ensamblado actual.
- protected internal, para un acceso limitado desde los objetos del ensamblado actual y para un acceso limitado a las clases heredadas.
- private, para que el método se utilice internamente o por una segunda instancia de un objeto del mismo tipo.

Si no se define ningún atributo, el método será privado.

Los modificadores opcionales

Después del atributo de visibilidad, se puede definir un modificador.

El modificador static declara el método como unido a un tipo y no a un objeto. Ya se ha abordado esta noción para los atributos y es el mismo principio para los métodos. Generalmente, el desarrollador agrupa en un juego de métodos de tipo static las operaciones que no justifican la instanciación de un objeto pero que están agrupadas en una clase "temática". Por ejemplo, una clase Calcular podrá ofrecer un juego de métodos de tipo static recibiendo los valores para sus operaciones como argumentos y devolviendo directamente el resultado. No es necesario ningún atributo para la operación y los métodos se agrupan en una clase cuyo nombre ya se ha mencionado: Calcular.

Desde la versión 6 de C# es posible importar en un archivo de código fuente solo los miembros de tipo static de las clases de un ensamblado.

```
using static MiEnsamblado;
```

Una clase también puede mezclar métodos "dinámicos", es decir, unidos a una instancia, y métodos de tipo static. Los métodos dinámicos pueden acceder a los miembros de tipo static,

pero no a la inversa. Los métodos de tipo static solo podrán utilizar los miembros de tipo static de la clase.

Otros modificadores propuestos por C# influyen en las reglas de herencia. Los modificadores virtual, abstract y sealed se utilizan del lado de las clases básicas, mientras que override y new se utilizan en el lado de las clases heredadas. Estos modificadores se estudiarán en el capítulo dedicado a la herencia.

Para terminar, el modificador async introducido con Visual Studio 2012 forma parte de los modos de uso avanzados de C# y se presentará en el capítulo El multithreading. Por el momento, es suficiente con saber que permite a un método precedido por este modificador un funcionamiento llamado "asíncrono", es decir, que no bloquea el hilo de ejecución, incluso si su operación es consecuente. Esta nueva funcionalidad es parecida al uso de los background threads, que se presentarán más adelante.

El tipo de retorno

Después de un eventual modificador o atributo de visibilidad, se define el tipo de retorno del método. Este retorno puede ser:

- La palabra clave void, cuando el método no devuelve nada.
- Un tipo perteneciente a la familia Valor (int por ejemplo).
- Un tipo perteneciente a la familia Referencia.

En este último caso, un objeto instanciado en el cuerpo del método puede sobrevivir a la ejecución del mismo (llave cerrada del método), si su referencia se devuelve para ser copiada y explotada de nuevo por el código que lo invoca.

El nombre del método

A continuación se declara el nombre del método. Las reglas son las mismas que para los nombres de los atributos, es decir, empezar con una letra o guión bajo (_). A continuación puede contener letras, cifras y o guiones bajos. Evite utilizar caracteres acentuados y, si su definición contiene varias palabras, utilice la notación de tipo camello (CamelCase), por ejemplo, MostrarColeccion. De una manera general, seleccione nombres de métodos explícitos, que hagan referencia al papel que tienen en su clase. Los nombres largos no son un impedimento, gracias al asistente a la escritura. Por tanto, evite nombres de tipo Funcion1 o MetodoBis...

Los argumentos del método

Después del nombre se declaran, entre paréntesis, los argumentos del método. Incluso si el método no tiene ningún argumento, hay que añadir un paréntesis abierto y otro cerrado.

```
bool ExportarContabilidad()
{//...}
```

Si el método recibe argumentos, se definen después del paréntesis abierto, en forma de lista de parejas tipo/nombre separadas por comas. Si el método solo utiliza un argumento pero hay que dejarlo para no cambiar los programas que le llaman, el nombre no es obligatorio.

```
int CalcularDescuento(int codigoCliente, int precioBase, int)
{//...}
```

Durante el uso del método, el código que lo invoca le pasará sus argumentos. El tipo de sus argumentos se deberá corresponder con los tipos de los argumentos esperados por el método. Los nombres, por el contrario, podrán ser diferentes, como se puede comprobar en el siguiente fragmento de código con un double radio y un double r como argumentos.

```
    class Calcular
    {
        // Método cálculo del círculo
        public double Perimetro(double r)
        {
            return 3.14 * r * 2;
        }
    }

    static void Main(string[] args)
    {
        Calcular c = new Calcular();

        // Calcular el perímetro de un círculo
        double radio = 2.3;
        double perimetro = c.Perimetro(radio);
    }
```

Por defecto, los argumentos se pasan como "valor", es decir, se realiza una copia del argumento durante la llamada al método. Para que el método pueda "trabajar" con los argumentos originales, las palabras clave ref u out deben preceder al tipo y de esta manera configurar un proceso de paso como "referencia". En este caso, el argumento se pasa en modo entrada-salida. Es importante entender las diferencias entre estos dos modos de paso de argumentos; volveremos sobre esto un poco más adelante.

Las instrucciones del método

Después del paréntesis que cierra la definición de los argumentos, empieza la implementación del código. Esta vez, es una llave abierta la que prefija la operación y una llave cerrada la que la termina. Cada línea de la operación se debe finalizar por un punto y coma.

b. Paso de argumentos por valor y por referencia

Como acabamos de presentar, existen dos modos de pasar argumentos a un método: por valor (el más frecuente) o por referencia.

Normalmente hay confusión entre el modo por valor o por referencia y el tipo de los argumentos, que pueden ser de la familia de los valores o de la familia de las referencias. Por ejemplo, un argumento de la familia de los valores, como un int, se puede pasar como argumento por valor o por referencia.

Para entender las diferencias, vamos a estudiar todos los casos posibles.

Pasar por valor un argumento de tipo valor

Este paso por valor, funciona realizando una copia del argumento en una variable local del método.

El siguiente fragmento de código muestra el paso por valor de un entero.

```
using System;
namespace DemoValRef
{
  class Program
  {
    static void Main(string[] args)
    {   // El Main instancia un objeto de tipo Demo
        // y llama a su método Execute
        Demo d = new Demo();
        d.Execute();
    }

    class Demo
    {
      public void Execute()
      {
        PruebaConLosValores t = new PruebaConLosValores();
        int contador = 10;
        Console.WriteLine("contador antes de la llamada: "
            + contador);
        // Durante la llamada a PasarValorPorValor,
        // se realizada una copia de contador
        t.PasarValorPorValor(contador);
```

```
        // Aquí, contador siempre vale 10
        // y la copia ha desaparecido
        Console.WriteLine("contador después de la llamada: "
            + contador);
      }
    }

    class PruebaConLosValores
    {
      // La variable i es local al método
      // Es la copia de contador
      public void PasarValorPorValor(int i)
      {
        // En la entrada al método, i vale 10
        Console.WriteLine("i en la entrada al método: "
            + i);
        // Se puede modificar sin afectar a contador
        i = 0;
        Console.WriteLine("i a la salida del método: "
            + i);
      }   // i va a desaparecer aquí... snif
    }
  }
}
```

Salida por la consola:

```
contador antes de la llamada: 10
i en la entrada al método: 10
i a la salida del método: 0
contador después de la llamada: 10
```

```
Pulse una tecla para continuar...
```

La copia de un tipo valor genera otro valor, es decir, una nueva entrada en la memoria de tipo pila (stack). Las modificaciones aportadas no afectan al valor inicial.

Para modificar el original, una primera solución consistiría en devolver la copia desde el método y después sustituir el valor original en el cuerpo principal, como en el siguiente código.

```
using System;

namespace DemoValRef
{
    class Program
    {
        static void Main(string[] args)
        {   // El Main instancia un objeto de tipo Demo
            // y llama a su método Execute
            Demo d = new Demo();
            d.Execute();
        }

        class Demo
        {
            public void Execute()
            {
                PruebaConLosValores t = new PruebaConLosValores();
                int contador = 10;
                Console.WriteLine("contador antes de la llamada: "
                    + contador);
                // Durante del llamada a PasarValorPorValor,
                // se realizada una copia de contador
                // Durante el retorno del método, el contenido de
                // contador se modifica
                contador = t.PasarValorPorValor(contador);
                // Aquí, contador vale ahora 0 y la
                // copia ha desaparecido
                Console.WriteLine("contador después de la llamada: "
```

```
                    + contador);
            }
        }

        class PruebaConLosValores
        {
            // La variable i es local al método
            // Es la copia de contador
            public int PasarValorPorValor(int i)
            {
                // A la entrada al método, i vale 10
                Console.WriteLine("i en la entrada al método: "
                    + i);
                // Se puede modificar sin afectar a
                // contador por el momento
                i = 0;
                Console.WriteLine("i a la salida del método: "
                    + i);
                // ahora se devuelve i modificado
                return i;
            }   // i va a desparecer aquí
        }
    }
}
```

Salida por la consola:

```
contador antes llamada: 10
i en la entrada al método: 10
i a la salida del método: 0
contador después de la llamada: 0
```

```
Pulse una tecla para continuar...
```

Esta solución es la más utilizada cuando el método solo debe devolver un argumento. Si, por ejemplo, desea devolver el resultado de un cálculo y un código de error, esta sintaxis no es conveniente.

La segunda solución consiste en pasar el valor por referencia.

Pasar un argumento de tipo valor por referencia

Como hemos visto, un método solo puede devolver un valor al mismo tiempo. Imaginemos un caso dónde el método deba devolver un código de error, como el valor true si el cálculo es correcto o false en caso contrario. En este caso es delicado devolver también el valor modificado.

Se puede cambiar nuestro método para pasar el argumento de cálculo por referencia y, de esta manera, autorizar al método para que trabaje directamente sobre el dato original del llamador. Aquí entra en escena la palabra clave ref que, situada justo antes del tipo del argumento, va a permitir este funcionamiento.

Sintaxis de un método que recibe un argumento en modo referencia:

```
[...]<tipo de retorno><Nombre>(ref [tipo] [argumento]){...}
```

Ejemplo:

```
public bool PasarValorPorReferencia(ref int i){...}
```

Para que el llamador sea consciente de que va a dar acceso directo (y por tanto sin control) a una de estas variables, el compilador le obliga a preceder su argumento de la palabra clave ref. En caso contario devuelve un error:

```
El argumento 1 se debe pasar con la palabra clave ’ref’
```

Ejemplo de llamada pasando un argumento por referencia:

```
t.PasarValorPorReferencia(ref contador);
```

A continuación se muestra el fragmento de código que presenta este nuevo principio:

```
using System;

namespace DemoValRef

{

  class Program
```

```
{
  static void Main(string[] args)
  {
    // El Main instancia un objeto de tipo Demo
    // y llama a su método Execute
    Demo d = new Demo();
    d.Execute();
  }
  class Demo
  {
      public void Execute()
      {
        PruebaConLosValores t = new PruebaConLosValores();
        int contador = 10;
        Console.WriteLine("contador antes de la llamada: "
            + contador);
        // Durante la llamada a PasarValorPorReferencia,
        // se crea una referencia a contador
        bool isOK = t.PasarValorPorReferencia(ref contador);
        // Aquí, contador vale ahora 0 gracias
        // a la palabra clave ref, el método que invoca
        // ha podido modificarlo directamente
        Console.WriteLine("contador después de la llamada: "
            + contador);
      }
  }
  class PruebaConLosValores
  {
    // La variable i corresponde a contador
    public bool PasarValorPorReferencia(ref int i)
    {
      // En la entrada al método, i vale 10
      Console.WriteLine(
```

```
          "i en la entrada al método: " + i);
        i = 0;
        Console.WriteLine(
          "i a la salida del método: "+ i);
        // El cálculo termina bien y devuelve true
        return true;
      }
    }
  }
}
```

Salida por la consola:

```
contador antes de la llamada: 10
i en la entrada al método: 10
i a la salida del método: 0
contador después de la llamada: 0
Pulse una tecla para continuar...
```

Pasar un argumento de tipo referencia por valor

Un objeto de tipo referencia tiene su propia zona en memoria en el heap, que contiene sus miembros y una referencia a esta zona registrada en una variable tipada dentro de la pila (stack).

Cuando se ejecuta:

```
MiClase mc = new MiClase();
```

Tiene dos partes:

- Un objeto de tipo MiClase que está en el heap.
- La variable mc, esta vez escrita en el stack, que referencia a la instancia del objeto MiClase del heap.

El desarrollador C++ hará la analogía con el bloque asignado en memoria y su puntero.

Como hemos visto con anterioridad, pasar un argumento por valor implica una copia del argumento. Si el método espera un objeto de tipo Referencia, el mecanismo de llamada va a duplicar esta referencia y la pasa al método. Esta referencia duplicada apunta al mismo objeto, el método llamador tendrá acceso al objeto original. El siguiente fragmento de código ilustra esta explicación.

```
using System;
```

```
namespace DemoValRef
{
  class Program
  {
    static void Main(string[] args)
    {
      // El Main instancia un objeto de tipo Demo
      // y llama a su método Execute
      Demo d = new Demo();
      d.Execute();
    }
    class MiClase
    {
      public int MiInt { get; set; }
      // Muestra la referencia en el heap
      // y el valor de la propiedad
      public override string ToString()
      {
          return "Ref:" + this.GetHashCode()
              + " MiInt:" + this.MiInt;
      }
    }
    class Demo
    {
      public void Execute()
      {
        PruebaConLosRef t = new PruebaConLosRef();

        // Instanciación de un objeto MiClase
        // inicializando su propiedad MiInt a 10
        MiClase mc = new MiClase() { MiInt = 10 };
        // Mostrar la consola
        Console.WriteLine(
```

```
          "mc antes de la llamada: "+ mc.ToString());
        // Durante la llamada a PasarRefPorValor, se crea
        // una copia de la referencia mc
        // en el stack
        t.PasarRefPorValor(mc);
        // Aquí mc.MiInt vale ahora 0
        // porque la copia de mc hace referencia al
        // mismo objeto en el heap.
        Console.WriteLine(
          "mc después de la llamada: " + mc.ToString());
      }
    }
    class PruebaConLosRef
    {
      // La variable m es local al método
      // y es una copia de mc
      public void PasarRefPorValor(MiClase m)
      {
        // En la entrada al método m.MiInt vale 10
        Console.WriteLine(
          "m en la entrada al método: " + m.ToString());
        m.MiInt = 0;
        Console.WriteLine(
          "m a la salida del método: " + m.ToString());
      }
    }
  }
}
```

Salida por la consola:

```
mc antes de la llamada: Ref:46104728 MiInt:10
m en la entrada al método: Ref:46104728 MiInt:10
m a la salida del método: Ref:46104728 MiInt:0
mc después de la llamada: Ref:46104728 MiInt:0
```

```
Pulse una tecla para continuar...
```

Observe que Object.GetHashCode() devuelve la referencia al objeto en memoria heap. Durante todos los intercambios, esta referencia no cambia (Ref:46104728), lo que prueba que se accede a la misma zona de memoria.

Pasar un argumento de tipo referencia por referencia

Existe una última combinación posible: pasar una referencia por referencia. No se utiliza con frecuencia. Entonces, ¿para qué puede servir?

Los desarrolladores C++ tendrán una idea, al hacer la analogía con el famoso puntero de puntero.

El método llamador recibe "una referencia a una referencia", puede reasignar un nuevo objeto en el heap y actualizar la referencia inicial al nuevo objeto.

El siguiente fragmento de código ilustra esta acción.

```
using System;

namespace DemoValRef
{
  class Program
  {
    static void Main(string[] args)
    {
      // El Main instancia un objeto de tipo Demo
      // y llama a su método Execute
      Demo d = new Demo();
      d.Execute();
    }

    class MiClase
    {
      public int MiInt { get; set; }

      // Muestra la referencia del heap
      // y el valor de la propiedad
      public override string ToString()
      {
          return "Ref:" + this.GetHashCode()
```

```
            + " MiInt:" + this.MiInt;
    }
}

class Demo
{
    public void Execute()
    {
        PruebaConLosRef t = new PruebaConLosRef();

        // Instanciación de un objeto MiClase,
        // inicializando su propiedad MiInt a 10
        MiClase mc = new MiClase() { MiInt = 10 };
        // Mostrar la consola
        Console.WriteLine("mc antes de la llamada: "
            + mc.ToString());

        // Durante la llamada a PasarRefPorRef, se ha creado
        // una referencia a la referencia mc
        t.PasarRefPorRef(ref mc);

        // Aquí, mc se corresponde con un
        // nuevo objeto en el heap.
        Console.WriteLine("mc después de la llamada: "
            + mc.ToString());
    }
}

class PruebaConLosRef
{
    // La variable m es local al método
    // y es una referencia en mc
    public void PasarRefPorRef(ref MiClase m)
```

```
      {
        // En la entrada al método, m.MiInt vale 10
        Console.WriteLine("m en la entrada al método: "
            + m.ToString());

        // Reasignación de un objeto MiClase con
        // actualización de su propiedad a 200
        m = new MiClase() { MiInt = 200 };

        Console.WriteLine("m a la salida del método: "
            + m.ToString());
      }
    }
  }
}
```

Salida por la consola:

```
mc antes de la llamada: Ref:46104728 MiInt:10
m en la entrada al método: Ref:46104728 MiInt:10
m a la salida del método: Ref:12289376 MiInt:200
mc después de la llamada: Ref:12289376 MiInt:200
Pulse una tecla para continuar...
```

Object.GetHashCode() devuelve ahora referencias diferentes. El método llamador ha modificado la zona de memoria del heap (Ref:46104728 que pasa a Ref:12289376).

¿Qué sucede con el objeto inicial? Si el objeto inicial solo tenía una referencia, como es nuestro caso, el garbage collector lo destruye. En caso contrario, sigue existiendo mientras que sea referenciado. El desarrollador C++ sabrá apreciar el trabajo del garbage collector, que simplifica mucho la gestión de la memoria...

En "referencia sobre referencia", C# también ofrece la palabra clave out. Si un argumento utiliza como prefijo la palabra clave out, el método llamador será responsable de su instanciación antes de la utilizarlo.

El siguiente fragmento de código ilustra esta definición.

```
using System;
namespace DemoValRef
{
```

```
class Program
{
  static void Main(string[] args)
  {
    // El Main instancia un objeto de tipo Demo
    // y llama a su método Execute
    Demo d = new Demo();
    d.Execute();
  }

  class MiClase
  {
    public int MiInt { get; set; }

    // Muestra la referencia del heap
    // y el valor de la propiedad
    public override string ToString()
    {
      return "Ref:" + this.GetHashCode()
          + " MiInt:" + this.MiInt;
    }
  }

  class Demo
  {
    public void Execute()
    {
      PruebaConLosRef t = new PruebaConLosRef();

      // Declaración de una referencia "null"
      MiClase mc = null;
      Console.WriteLine("mc antes de la llamada=null");
      // Durante la llamada a PasaRefPorOut, se crea
```

```
        // una referencia a la referencia mc
        t.PasaRefPorOut(out mc);
        // Aquí, mc.MiInt vale ahora 200
        // porque PasaRefPorOut ha creado un
        // objeto en el heap que contiene 200
        Console.WriteLine("mc después de la llamada: "
            + mc.ToString());
      }
    }
    class PruebaConLosRef
    {
      // La variable m es local al método
      // y es una copia de mc
      public void PasaRefPorOut(out MiClase m)
      {
        // En la entrada, m es null
        Console.WriteLine("m en la entrada =null");
        // Asignación de un objeto MiClase con
        // actualización de su propiedad a 200
        m = new MiClase() { MiInt = 200 };

        Console.WriteLine("m a la salida del método: "
            + m.ToString());
      }
    }
  }
}
```

Salida por la consola:

```
mc antes de la llamada=null
m en la entrada =null
m a la salida del método: Ref:46104728 MiInt:200
mc después de la llamada: Ref:46104728 MiInt:200
```

```
Pulse una tecla para continuar...
```

De nuevo, el compilador obliga a utilizar la palabra clave out cuando se llama al método.

Precauciones durante el uso

Por definición, una referencia era "nullable", por lo que es necesario probar su instancia antes de acceder a sus miembros. Estas precauciones hacen un poco pesado el código fuente.

Desde C# 6, una nueva sintaxis evita que estas las pruebas tengan varias líneas, insertando el carácter ? entre el nombre de la instancia y el miembro.

Si el miembro es un atributo y la instancia del objeto es null, entonces el resultado devuelto será null.

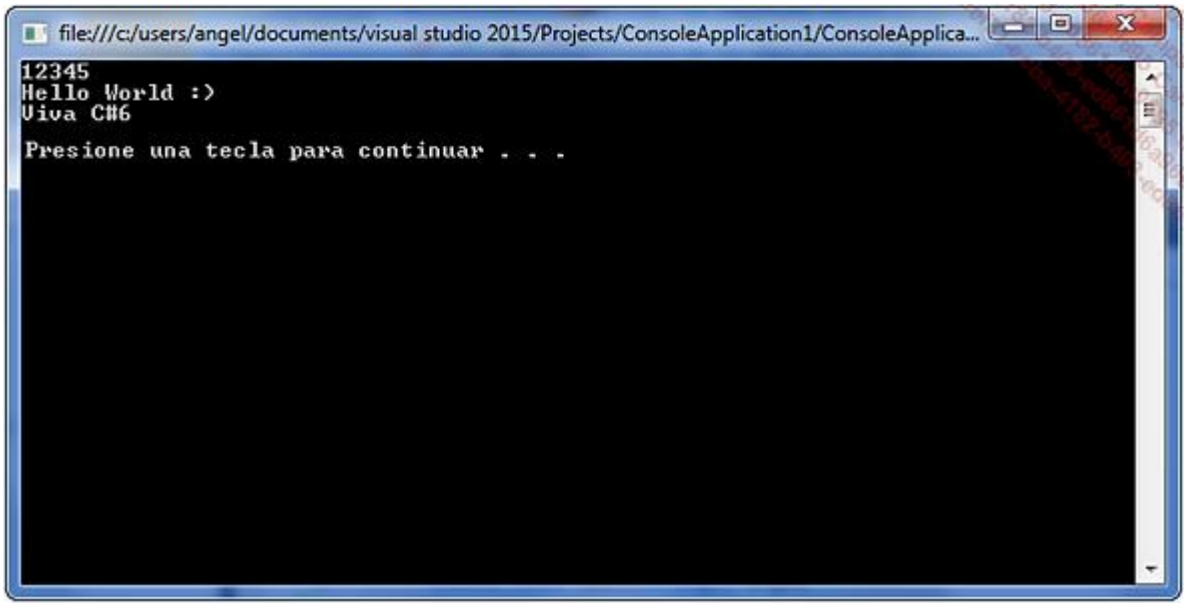
Si el miembro es un método y la instancia del objeto es null, entonces el método no se llamará.

El siguiente código muestra el uso de esta sintaxis con el método Accion.

```
class Prueba
{
    public int PropiedadTipoValor;
    public string PropiedadTipoReferencia;
    public void Tratamiento(string argumento)
    {
        Console.WriteLine(argumento);
    }
}

class Program
{
    static void Accion (Prueba origen)
    {
        int? v = origen?.PropiedadTipoValor;
        Console.WriteLine(v);

        string s = origen?.PropiedadTipoReferencia;
        Console.WriteLine(s);
```

```
        origen?.Tratamiento("Viva C#6");


    }
    static void Main(string[] args)
    {
        Prueba c = new Prueba();
        c.PropiedadTipoValor = 12345;
        c.PropiedadTipoReferencia = "Hello World :)";
        Accion(c);

        Accion (null);
    }
}
```

La salida por la consola es:



Observe que con esta sintaxis la recuperación de un atributo de tipo valor solo se podrá realizar sobre un valor de tipo nullable. Si esto representa un problema, habría que utilizar los caracteres ??, que permiten definir un valor por defecto en caso de nulabilidad.

```
int k = origen?.PropiedadTipoValor?? 3;
```

En este caso:

- si origen es null, entonces k valdrá 3.
- si origen es diferente de null, k valdrá el atributo PropiedadTipoValor de origen.

9. Mecanismo de las excepciones

a. Presentación

Las anomalías que el runtime (CLR) detecta durante la ejecución se envían a la aplicación por medio de un mecanismo común a todos los lenguajes de .NET: las excepciones.

Por ejemplo, si ejecuta el siguiente código, el runtime va a "enviar" una excepción, porque el programa intenta escribir más allá del límite de la tabla (tab[5] = 3;).

```
namespace DemoExcepcion
{
    class Program
    {
        static void Main(string[] args)
        {
           int[] tab = new int[5];
            tab[5] = 3;      // excepción porque el índice va de 0 a 4
        }
    }
}
```

Dando por hecho que el código no está previsto para tratar las excepciones, un cuadro de diálogo informa al usuario de que ha aparecido un problema no gestionado.

El mensaje es muy preciso; indica que la excepción es de tipo System.IndexOutOfRangeException y que se produjo como consecuencia del uso de un índice fuera de los límites de la tabla. En este caso se trata de un bloqueo de codificación.

Hay otras fuentes de error, como las entradas erróneas. Por ejemplo, el programa espera una cifra y el usuario introduce una serie de letras, pero no se hace ninguna operación de comprobación, por lo que se produce un error de conversión.

También hay errores que da el propio "sistema". Por ejemplo, su programa intenta escribir un archivo en una llave USB de memoria que el usuario desconecta antes de que termine el trabajo.

El mecanismo de las excepciones se ejecuta para estos tres tipos de error. Naturalmente, el desarrollador puede evitar las dos primeras fuentes de error. Sin embargo, si persisten los fallos, se producirán excepciones.

Las excepciones no están reservadas a la CLR. Sus clases también podrán enviar excepciones y, lo que es más significativo, excepciones específicas de su código.

Devolver un código de error en un método no obliga al desarrollador a probarlo; enviar una excepción en un método obliga al desarrollador a tratarla.

Normalmente, un método puede obtener resultados diferentes si devuelve un código de error, que permite distinguirlo. Por ejemplo, un método de apertura de archivo devuelve información diferente según el resultado de su ejecución. Posteriormente, nada obliga al desarrollador a probar este código de retorno. Se puede ejecutar una operación de lectura cuando el archivo no se abre. Con el mecanismo de las excepciones, el error bloquea la ejecución del programa, si el desarrollador no ha previsto un operación ad hoc.

b. Principio de funcionamiento de las excepciones

Hay que distinguir la parte que la emite del tratamiento de la excepción.

Parte de la emisión

- Un método se está ejecutando.
- Se detecta un funcionamiento incorrecto.
- El método asigna un objeto de tipo (o que hereda de) System.Exception.
- El método informa los atributos del objeto exception para que el origen del error sea lo más explícito posible.
- El método envía el error utilizando la palabra clave throw.
- La ejecución del método se interrumpe inmediatamente para devolver el control al código que invoca.
- El mecanismo busca en la pila de llamadas aquella que ha provocado en la ejecución de este método una operación de excepción.
- Si en la pila de llamadas se encuentra una operación ad hoc, se ejecuta.
- En otro caso, la CLR detiene bruscamente la ejecución y muestra un cuadro de diálogo explicando que la excepción no está gestionada.

Ejemplo de código:

```
using System;


namespace DemoExcepcion
{
  class Program
```

```
{
  static void Main(string[] args)
  {
    // El Main instancia un objeto de tipo Demo
    // y llama a su método Execute
    Console.WriteLine("Inicio de Main");
    Demo d = new Demo();
    d.Execute();
    Console.WriteLine("Fin de Main");
  }
}

class Demo
{
  public void Execute()
  {
    Console.WriteLine("Inicio de Execute");
    MiClase mc = new MiClase();
    mc. MiMetodo(false);
    Console.WriteLine("Fin de Execute");
  }
}

class MiClase
{
  // Este método devuelve una excepción
  // "bajo petición" para permitirnos
  // entender el mecanismo
  public void MiMetodo(bool SimulaError)
  {
    Console.WriteLine("Inicio de MiMetodo");
    if (SimulaError == true)
    {
```

```
        // provoca una excepción "general"
        throw new Exception("Error de MiMetodo");
        //la ejecución se detiene inmediatamente
        Console.WriteLine("Nunca se llega a esta línea");
      }
      Console.WriteLine("Fin de MiMetodo");
    }
  }
}
```

En este ejemplo, la clase Demo instancia y utiliza el método MiMetodo de la clase MiClase. Para que podamos estudiar el mecanismo, el método genera una excepción cuando se le invoca pasando true como argumento.

El código que permite generar la excepción es el siguiente:

```
throw new Exception("Error de MiMetodo");
```

Este código utiliza la sobrecarga del constructor de la clase de base System.Exception para transmitir un mensaje explicativo al código que lo invoca.

En la actualidad, la ejecución del programa provoca la siguiente salida por la consola:

```
Inicio de Main
Inicio de Execute
Inicio de MiMetodo
Fin de MiMetodo
Fin de Execute
Fin de Main
Pulse una tecla para continuar...
```

Supongamos que la llamada al método MiMetodo se realiza con true como argumento.

```
mc.MiMetodo(true);
```

La salida por la consola se convierte en la siguiente:

```
Inicio de Main
Inicio de Execute
Inicio de MiMetodo
```

```
Excepción no controlada: System.Exception: Error de MiMetodo
   en DemoExcepcion.MiClase.MiMetodo(Boolean SimulaError)
en DemoException\Program.cs:línea 97
   en DemoExcepcion.Demo.Execute()
en DemoException\Program.cs:línea 81
   en DemoExcepcion.Program.Main(String[] args)
en DemoException\Program.cs:línea 70
Pulse una tecla para continuar...
```

El programa no se cierra correctamente. Es normal porque el código no está preparado para tratar la excepción y, por tanto, el error se devuelve hasta el primer nivel.

Por tanto, la sintaxis que permite generar una excepción es:

```
throw <objeto de tipo o heredado de System.Exception>;
```

Parte del tratamiento

- El código que utiliza el método se diseñó para responder a las excepciones.
- Tan pronto como se produce la excepción, la ejecución del código se envía a la parte diseñada para su tratamiento.

Se ha modificado el código anterior para integrar el tratamiento de la excepción.

```
using System;


namespace DemoExcepcion
{
  class Program
  {
    static void Main(string[] args)
    {
      // El Main instancia un objeto de tipo Demo
      // y llama a su método Execute
      Console.WriteLine("Inicio de Main");
      Demo d = new Demo();
      d.Execute();
      Console.WriteLine("Fin de Main");
    }
```

```
}

class Demo
{
  public void Execute()
  {
    Console.WriteLine("Inicio de Execute");
    MiClase mc = new MiClase();
    try
    {
      // Sección de código de monitorización
      mc.MiMetodo(false);
      mc.MiMetodo(true);
      Console.WriteLine("Fin del bloque try");
    }
    catch (Exception e)
    {
      // Código que se llama cuando
      // se genera la excepción
      Console.WriteLine("Excepción detectada");
      Console.WriteLine("Motivo: " + e.Message);
    }
    Console.WriteLine("Fin de Execute");
  }
}



class MiClase
{
  // Este método genera una excepción
  // "bajo petición" para permitirnos
  // entender el mecanismo
  public void MiMetodo(bool SimulaError)
```

```
    {
      Console.WriteLine("Inicio de MiMetodo");
      if (SimulaError == true)
      {
        // se genera una excepción "general"
        throw new Exception("Error de MiMetodo");
        // la ejecución se detiene inmediatamente
        Console.WriteLine("Nunca se llega a esta línea");
      }
      Console.WriteLine("Fin de MiMetodo");
    }
  }
}
```

La sintaxis mínima para tratar una excepción está formada por dos partes.

- La primera consiste en enmarcar en un bloque try las instrucciones susceptibles de generar la excepción.
- La segunda ofrece un bloque catch que recibe como argumento el tipo de la excepción "capturada".

A continuación se muestra la nueva salida por la consola:

```
Inicio de Main
Inicio de Execute
Inicio de MiMetodo
Fin de MiMetodo
Inicio de MiMetodo
Excepción detectada
Motivo: Error de MiMetodo
Fin de Execute
Fin de Main
Pulse una tecla para continuar...
```

La excepción se ha capturado correctamente y la clase Demo conserva el control de la ejecución del programa, que llega hasta el final y se cierra sin error.

Observe que la línea Console.WriteLine("Fin del bloque try"); no se ha ejecutado porque la operación se desvió automáticamente al bloque catch.

Por el contrario, observe que la línea Console.WriteLine("Fin de Execute"); se ha ejecutado después el bloque catch.

Algunas veces, el bloque catch no puede tratar la excepción completamente. En este caso, es mejor enviarla al nivel superior (en nuestro ejemplo: el Main). Para esto, es suficiente con ejecutar la instrucción throw;.

A continuación se muestra el código modificado en consecuencia:

```
using System;

namespace DemoExcepcion
{
  class Program
  {
    static void Main(string[] args)
    {
      // El Main instancia un objeto de tipo Demo
      // y llama a su método Execute
      Console.WriteLine("Inicio de Main");
      try
      {
        Demo d = new Demo();
        d.Execute();
      }
      catch (Exception e)
      {
        Console.WriteLine("Excepción en Main");
        Console.WriteLine("Motivo: " + e.Message);
      }
      Console.WriteLine("Fin de Main");
    }
  }


  class Demo
  {
    public void Execute()
    {
```

```
    Console.WriteLine("Inicio de Execute");
    MiClase mc = new MiClase();
    try
    {
      // Sección de código de monitorización
      mc.MiMetodo(false);
      mc.MiMetodo(true);
      Console.WriteLine("Fin del bloque try");
    }
    catch (Exception e)
    {
      // Código que se llama cuando
      // se genera la excepción
      // ...
      // La excepción no se puede tratar
      // aquí por un algún motivo
      // Se devuelve al "nivel superior"
      throw;
    }
    Console.WriteLine("Fin de Execute");
  }
}



class MiClase
{
  // Este método genera una excepción
  // "bajo demanda" para permitirnos
  // entender el mecanismo
  public void MiMetodo(bool SimulaError)
  {
    Console.WriteLine("Inicio de MiMetodo");
```

```
      if (SimulaError == true)
      {
        // se genera una excepción "general"
        throw new Exception("Error de MiMetodo");
        //... la ejecución se detiene inmediatamente
        Console.WriteLine("Nunca se llega a esta línea");
      }
      Console.WriteLine("Fin de MiMetodo");
    }
  }
}
```

La salida por la consola se convierte en:

```
Inicio de Main
Inicio de Execute
Inicio de MiMetodo
Fin de MiMetodo
Inicio de MiMetodo
Excepción en Main
Motivo: Error de MiMetodo
Fin de Main
Pulse una tecla para continuar...
```

El código de finally

En el código anterior, observamos que la línea Console.WriteLine("Fin de Execute"); no se ha ejecutado. Es normal, porque la instrucción throw ha detenido la operación "normal". En ocasiones, esto puede representar un problema cuando, por ejemplo, hay que ejecutar algunas instrucciones de "limpieza" en el objeto que ha generado la excepción.

Esta es la razón de ser del bloque finally{...}, cuyas líneas se ejecutan siempre, independientemente de las operaciones de try.

```
using System;


namespace DemoExcepcion
{
  class Program
```

```
{
  static void Main(string[] args)
  {
    // El Main instancia un objeto de tipo Demo
    // y llama a su método Execute
    Console.WriteLine("Inicio de Main");
    try
    {
      Demo d = new Demo();
      d.Execute();
    }
    catch (Exception e)
    {
      Console.WriteLine("Excepción en Main");
      Console.WriteLine("Motivo: " + e.Message);
    }
    Console.WriteLine("Fin de Main");
  }
}

class Demo
{
  public void Execute()
  {
    Console.WriteLine("Inicio de Execute");
    MiClase mc = new MiClase();
    try
    {
      // Sección de código de monitorización
      mc.MiMetodo(false);
      mc.MiMetodo(true);
      Console.WriteLine("Fin del bloque try");
    }
```

```
    catch (Exception e)
    {
      // Código que se llama cuando
      // se genera la excepción
      // ...
      // La excepción no se puede tratar
      // aquí por algún motivo
      // Por tanto, se envía al "nivel superior"
      throw;
    }
    finally
    {
      Console.WriteLine("Fin de Execute");
      // Código llamado sistemáticamente
      mc.FinUtilizacion();
    }
  }
}

class MiClase
{
  // Este método genera una excepción
  // "bajo demanda" para permitirnos
  // entender el mecanismo
  public void MiMetodo(bool SimulaError)
  {
    Console.WriteLine("Inicio de MiMetodo");
    if (SimulaError == true)
    {
        // asigna y después genera una excepción "general"
        throw new Exception("Error de MiMetodo");
        //la ejecución se detiene inmediatamente
        Console.WriteLine("Nunca se llega a esta línea");
```

```
      }
      Console.WriteLine("Fin de MiMetodo");
    }

    public void FinUtilizacion()
    {
      Console.WriteLine("MiClase.FinUtilizacion");
    }
  }
}
```

La salida por la consola se convierte en:

```
Inicio de Main
Inicio de Execute
Inicio de MiMetodo
Fin de MiMetodo
Inicio de MiMetodo
Fin de Execute
MiClase.FinUtilizacion
Excepción en Main
Motivo: Error de MiMetodo
Fin de Main
Pulse una tecla para continuar...
```

Aquí se ve que, incluso si la excepción se trata en Main, el método FinUtilizacion del objeto MiClase se ha llamado igualmente gracias a finally.

Por tanto, la sintaxis del tratamiento de una excepción es:

```
try{ <bloque que pude presentar riesgos de generar una excepción>}
catch(<objeto tipo Exception> ){<tratamiento de la excepción>}
finally{<bloque que se ejecuta, haya o no excepción y se redirija o no la excepc
ión>}
```

c. Soporte a varias excepciones

En función del contenido del bloque try se pueden generar varios tipos de excepciones y, teóricamente, varios bloques catch para capturarlas. Afortunadamente, si la CLR no encuentra el tipo exacto de la excepción en sus catch buscará como vincularse con una operación menos específica; en la ocurrencia, busca una excepción "madre" de la que fue generada.

Por tanto, en particular, si escribe solo un catch de tipo Exception va a recuperar todo, porque Exception es la clase madre de todas las excepciones (ya sean de .NET o de sus ensamblados). Por el contrario, va a perder la precisión en su diagnóstico.

También es interesante tener información específica para reaccionar lo más eficazmente posible a la excepción y, para ello, puede codificar varios bloques catch, uno tras otro. El orden en la lista es importante: debe empezar a escribir los catch para tratar las excepciones más específicas, para abordar a continuación las más generalistas.

Ejemplo de catch múltiple:

```
try
{
  // operaciones;
}
catch (FormatException fe)
{
  // Acciones si FormatException
}
catch (OverflowException oe)
{
  // Acciones si OverflowException
}
catch (Exception e)
{
  // Acciones si Exception
}
```

d. try ... catch ... finally y using

Es legítimo preguntarse sobre las secciones de código ejecutado cuando se usan juntos using{… y la secuencia try ... catch ... finally.

En el siguiente código, una clase ClaseDePrueba ofrece un método Trabajo que genera una excepción. Su método Dispose se utiliza para que se pueda seguir la secuencia de las llamadas...

```
class ClaseDePrueba: IDisposable
{
    public void Trabajo()
    {
        Console.WriteLine("Método Trabajo de ClaseDePrueba");
        Console.WriteLine("ClaseDePrueba genera una excepción");
        throw new Exception(" de ClaseDePrueba");
    }

    public void Dispose()
    {
        Console.WriteLine("Método Dispose de ClaseDePrueba");
    }
}
```

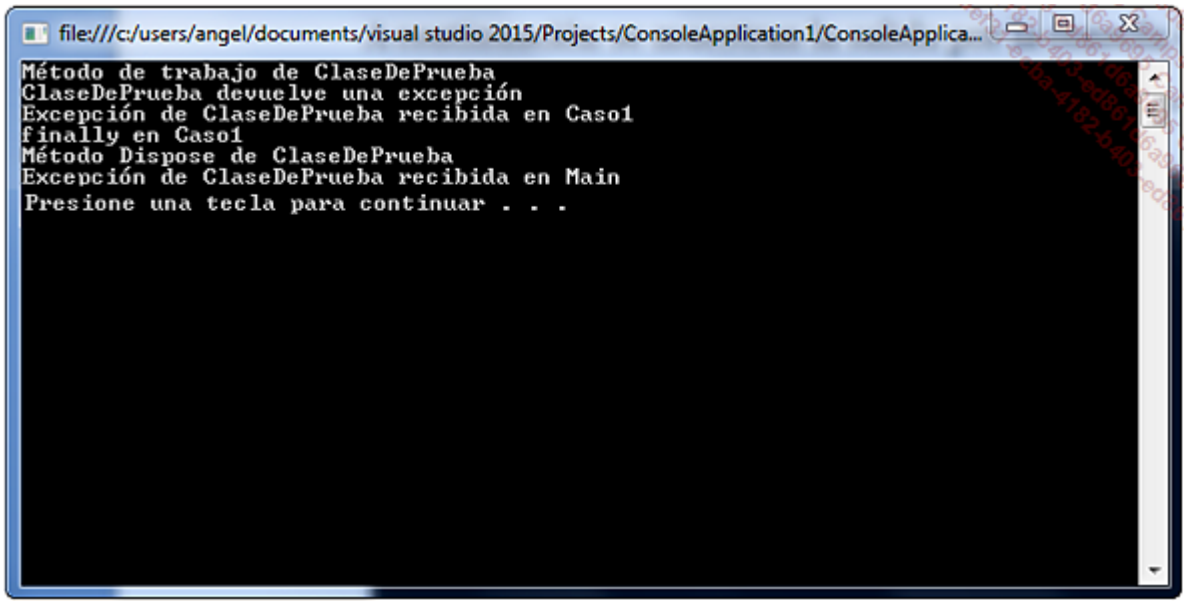

Premier caso de uso:

ClaseDePrueba se instancia ANTES que la secuencia try ... catch ... finally, utilizando una sintaxis using y la secuencia catch rechaza la excepción.

```
class Program
{
```

```
    static void Main(string[] args)
    {
        try
        {  Caso1();  }  catch (Exception e)  {
            Console.WriteLine("Excepción de " + e.Message +
" recibida en Main");
        }
    }
    static void Caso1()
    {
        using (ClaseDePrueba tc = new ClaseDePrueba())
        {
            try  {
                tc.Trabajo();
            }
            catch (Exception e)
            {
                Console.WriteLine("Excepción de " + e.Message +
" recibida en Caso1");
                throw;
            }
            finally
            {
                Console.WriteLine("finally en Caso1");
            }
        }
    }
}
```

A continuación se muestra la correspondiente salida por consola:



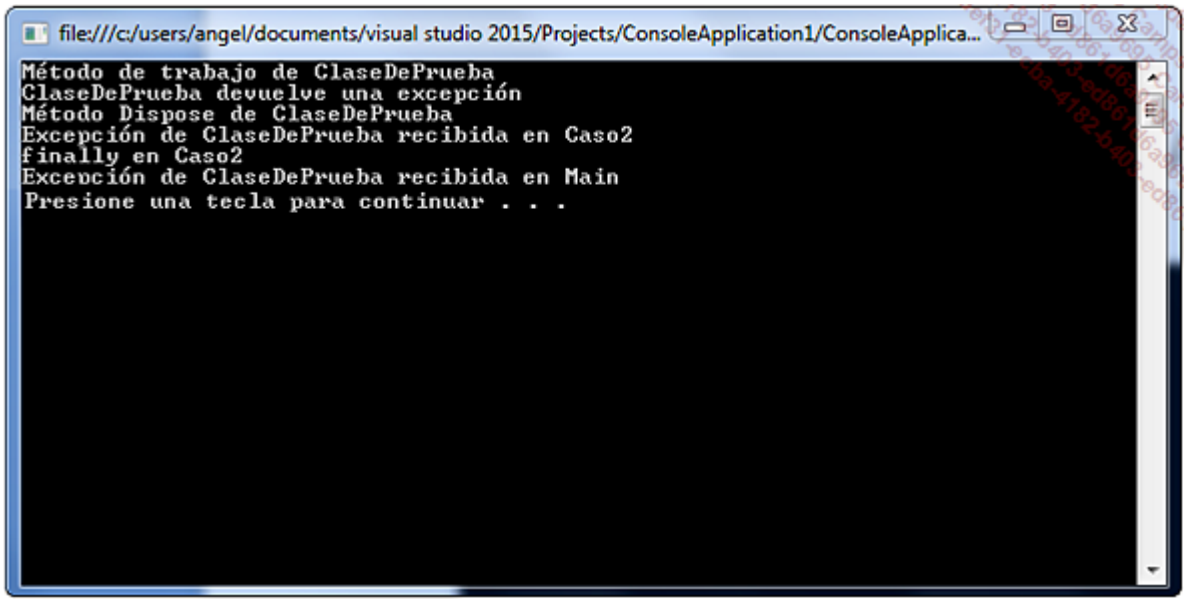
De manera lógica, podemos comprobar el encadenamiento de finally y de dispose: no se ha olvidado nada.

Segundo caso de uso:

ClaseDePrueba se instancia en la secuencia try, utilizando una sintaxis using y la secuencia catch rechaza la excepción.

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            Caso2();
        }
        catch (Exception e)
        {
            Console.WriteLine(
"Excepción de " + e.Message + " recibida en Main");
        }
    }
    static void Caso2()
    {
        try
```

```
        {
            using (ClaseDePrueba tc = new ClaseDePrueba())
            {
                tc.Trabajo();
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(
"Excepción de " + e.Message + " recibida en Caso2");
            throw;
        }
        finally
        {
            Console.WriteLine("finally en Caso2");
        }
    }
}
```



De nuevo, no se ha olvidado nada; el orden de llamada de dispose y finally sencillamente se ha modificado respecto a la sintaxis anterior.

10. Sobrecarga de los métodos

Como con sus constructores, una clase puede ofrecer varios métodos con el mismo nombre, pero con argumentos diferentes. Esta posibilidad permite adaptar el mismo "verbo" (nombre del método) a diferentes circunstancias.

El siguiente fragmento de código muestra un ejemplo de sobrecarga (overloading) del método Perimetro, que aquí se ha implementado en tres versiones.

```
using System;


namespace DemoSobrecargaMetodos
{
    class Program
    {
        class Calcular
        {   // Método para el rectángulo
            public double Perimetro(double longitud,
                double altura)
            {
                return 2 * longitud + 2 * altura;
            }
            // Método para el triángulo
            public double Perimetro(double lado1,
                double lado2, double lado3)
            {
                return lado1 + lado2 + lado3;
            }
            // Método para el círculo
            public double Perimetro(double radio)
            {
                return 3.14 * radio * 2;
            }
        }


        static void Main(string[] args)
        {
```

```
            Calcular c = new Calcular();
            double Perimetro = 0;

            // Cálculo del perímetro de un círculo
            double radio = 2.3;
            Perimetro = c.Perimetro(radio);
            // Cálculo del perímetro de un triángulo
            double c1 = 2, c2 = 10, c3 = 5;
            Perimetro = c.Perimetro(c1, c2, c3);
            // Cálculo del perímetro de un rectángulo
            double longitud = 10, altura = 13;
            Perimetro = c.Perimetro(longitud, altura);

            Console.ReadKey();
        }
    }
}
```

El compilador es el que deduce, en función de los argumentos que se pasan, el método que se debe invocar. Para esto, necesariamente hay que "firmarlos" como métodos diferentes. En el caso de la sobrecarga, la firma del método incluye su nombre, sus argumentos, pero no su tipo de retorno. Además, veremos que en el caso de los delegate, la firma contiene el código de retorno...

Por ejemplo, el siguiente fragmento de código es incorrecto.

```
    public double Perimetro(double radio)
    {
        return 3.14 * radio * 2;
    }

    public int Perimetro(double radio)
    {
        return (int)(3.14 * radio * 2);
    }
```

El compilador devuelve el error:

```
"El tipo 'DemoSobrecargaMetodos.Program.Calcular' ya ha definido un
miembro llamado 'Perimetro' con los mismos tipos de argumento."
```

No dude en sobrecargar sus métodos para enriquecer sus objetos, porque su programación será más intuitiva.

11. Ejercicio

a. Enunciado

Crear una solución con un proyecto de tipo consola.

Crear una clase Usuario que contenga las propiedades Nombre, Apellido y Edad, con las siguientes funcionalidades:

- Un constructor ad hoc debe simplificar la creación/inicialización de la clase.
- La edad introducida debe estar comprendida entre 0 y 109 años. Se debe informar al usuario de la clase, si la edad introducida no está en el rango permitido.
- El método System.Objet.ToString se debe sobrecargar para resumir el contenido de la clase.

En el programa principal (main), cree cuatro usuarios que contengan la siguiente información:

Nombre	Apellido	Edad
María	González-Aller	45
Ángel	Sánchez	50
José	La Riva	115
Mateo	Sánchez	28

A cada creación, le debe corresponder una línea de resumen del registro que se muestre en la consola.

Cualquier error de formato debe generar una línea de error en la consola y no bloquear el resto del programa. A continuación, se muestra la visualización que se persigue:

```
María González-Aller edad: 45

Ángel Sánchez edad: 51

José La Riva edad no real (-1)

Mateo Sánchez edad: 28
```

b. Consejos

Como es imposible devolver un código de error desde un constructor, habría que utilizar el mecanismo de las excepciones para comunicarse con el usuario de la clase cuando la edad no respete la restricción impuesta por las especificaciones.

El lugar ideal para realizar la prueba de validez de la edad es el método set de su propiedad.

Aunque existe una excepción de .NET llamada ArgumentOutOfRangeException, que es adecuada para devolver un error sobre la edad, se le sugiere crear un objeto original para entender mejor la codificación.

Como un error no debe bloquear el resto del programa, hay que externalizar la creación de las instancias Usuario en un método que gestione las excepciones.

c. Corrección

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;


namespace LabCtorYExcepciones

{

  class Program

  {

    // Este método permite fabricar las instancias

    // Usuario. Es de tipo static porque

    // se llama desde el método Main que

    // es de tipo static
```

```
  static Usuario AñadirUsuario(
    string nombre, string apellido, uint edad)
  {
    Usuario dest = null;
    // El try/catch va a actuar si los argumentos
    // de la instanciación son incorrectos.
    try
    {
      dest = new Usuario(nombre, apellido, edad);
      Console.WriteLine(dest.ToString());
    }
    catch (UsuarioExcepcion ce)
    {
      Console.WriteLine(
        ce.ErrorMessage + "(" + ce.ErrorCode + ")");
    }
    return dest;
  }

  static void Main(string[] args)
  {
    Usuario margonz =
      AñadirUsuario("María", "González-Aller", 45);
    Usuario angsan =
      AñadirUsuario("Ángel", "Sánchez", 51);
    Usuario matsan =
      AñadirUsuario("Mateo", "Sánchez", 28);
    Usuario joslri =
      AñadirUsuario("José", "La Riva", 115);

    Console.ReadKey();
  }
}
```

```
// La clase Usuario contiene tres propiedades
// Nombre, Apellido y Edad
// Solo la edad se valida en su set
class Usuario
{
  private String nombre;
  public String Nombre
  {
    get { return nombre; }
    set { nombre = value; }
  }

  private string apellido;
  public string Apellido
  {
    get { return apellido; }
    set { apellido = value; }
  }

  private uint edad;
  public uint Edad
  {
    get { return edad; }
    // Prueba de validez de la edad
    set
    {
      // Si la edad está comprendida en el rango
      // entonces se registra
      if (value >= 0 && value < 110)
        edad = value;
      else
      {
```

```
        // En caso contrario, se genera una excepción "custom"
        throw new UsuarioExcepcion(
          -1,
          string.Format("{0} {1} Edad {2} incorrecta",
            this.Nombre, this.Apellido, value));
      }
    }
  }
  public Usuario(string nombre, string apellido, uint edad)
  {
    this.Nombre = nombre;
    this.Apellido = apellido;
    this.Edad = edad;
  }

  // Se usa el método Object.ToString para resumir
  // el contenido del objeto
  public override string ToString()
  {
    return string.Format("{0} {1} edad: {2}",
      this.Nombre, this.Apellido, this.edad);
  }
}

// La clase UsuarioExcepcion extiende
// la clase Exception; es una obligación...
// Esta clase contiene un código error y un mensaje
class UsuarioExcepcion: Exception
{
  int errorCode;
  public int ErrorCode
  {
    get { return errorCode; }
```

```
      set { errorCode = value; }
    }

    string ErrorMessage;
    public string ErrorMessage
    {
      get { return ErrorMessage; }
      set { ErrorMessage = value; }
    }

    public UsuarioExcepcion(
      int errorCode, string ErrorMessage)
    {
      this.ErrorCode = errorCode;
      this.ErrorMessage = ErrorMessage;
    }
  }
}
```

Las interfaces

1. Introducción

Explicar las interfaces y su interés siempre es mejor con un ejemplo concreto que lo apoye. Por tanto, imagine un programa que permita controlar un sistema domótico desde un teléfono móvil (con nuestros smartphones siempre conectados, el interés por este tipo de aplicaciones va a sufrir una verdadera explosión). Este programa gráfico permitirá controlar las persianas eléctricas, leer la temperatura, encender el horno, en resumen leer y escribir los estados lógicos (verdadero o falso) y leer y escribir valores analógicos (00 a ff, por ejemplo).

En este tipo de aplicaciones hay que controlar que no se depende de un hardware concreto. Un cambio en la tarjeta de entrada/salida, es decir, la tarjeta que va a leer los sensores y controlar los relés bajo demanda debe afectar lo menos posible al código existente.

En esto nos van a ayudar las interfaces de programación.

2. El contrato

Para tener éxito en nuestra independencia respecto al hardware, hay que limitar sus relaciones a su forma más sencilla y "firmar un contrato".

Por analogía, se puede decir que gracias al "formato estándar del Conector Jack 3.5mm estéreo, cualquier par de auriculares se puede conectar a cualquier reproductor digital. Esta famosa toma juega el papel de interfaz entre dos hardwares que no son necesariamente del mismo fabricante.

Limitar las relaciones a su expresión más sencilla implica enumerar las funcionalidades mínimas esperadas por la tarjeta de entrada/salida. Esta lista es un tipo de contrato que el hardware deberá respetar obligatoriamente. Para volver a la analogía anterior, los fabricantes de auriculares ofrecen productos con conectores que tienen diámetros estandarizados y la conexión es posible gracias a esta "interfaz".

Entonces, para este proyecto, ¿cuáles son nuestras necesidades?

Hay que poder:

- Leer los estados binarios de las entradas referenciadas - interruptores, pulsadores y sensores de presencia.
- Leer de los valores analógicos de las entradas referenciadas - sensores de temperatura, sensores de luminosidad y sensores de sonido.
- Solicitar las salidas binarias referenciadas - persianas, horno, portal, etc.
- Solicitar las salidas analógicas referenciadas - adaptadores de luminosidad, etc.

A continuación se muestra la lista mínima de funciones que la encapsulación de una tarjeta de entrada/salida deberá ofrecer obligatoriamente para ser compatible con nuestra aplicación:

```
bool LeerBinario(uint numTarjeta, uint numEntrada);

int LeerAnalogica(uint numTarjeta, uint numEntrada);

void EscribirBinario(uint numTarjeta, uint numSalida, bool estado);

void EscribirAnalogica(uint numTarjeta, uint numSalida, int val);
```

Este contrato no contiene código ni datos.

3. Declaración de una interfaz

Sintaxis de declaración:

```
[Modificador de acceso] interfaz nombreInterfaz  [:Lista

interfaces de base]

{cuerpo de la interfaz}
```

- Modificador de acceso: opcional si la interfaz se declara en una clase, prohibido si la interfaz se declara en el primer nivel de un espacio de nombres.
- nombreInterfaz: en C#, es habitual utilizar el prefijo 'I' delante del nombre de las interfaces (por ejemplo, IBasicIO). Es solo una convención pero se utiliza mucho.
- Lista de interfaces de base: opcional; esta lista define la o las interfaces de la que hereda la nueva interfaz. Por tanto, la clase que implemente esta nueva interfaz deberá implementar todos los miembros de todas las interfaces de la lista.

Ahora podemos "envolver" nuestra lista de firmas de métodos en una **interfaz** que vamos a llamar, por ejemplo, IBasicIO.

```
// Interfaz que contiene los métodos a soportar obligatoriamente
public interface IBasicIO
{
  // Lectura de una entrada binaria
  // referenciar la tarjeta + número de entrada
  bool LeerBinario(uint numTarjeta, uint numEntrada);
  // Lectura de una entrada analógica
  // referenciar la tarjeta + número de entrada
  int LeerAnalogica(uint numTarjeta, uint numEntrada);
  // Escritura de una salida binaria
  // referenciar la tarjeta + número de salida
  void EscribirBinario(uint numTarjeta, uint numSalida, bool estado);
  // Escritura de una salida analógica
  // referenciar la tarjeta + número de salida
  void EscribirAnalogica(uint numTarjeta, uint numSalida, int val);
}
```

Esta interfaz constituye el contrato que cualquier tarjeta deberá respetar para funcionar con nuestra aplicación. Parece una clase que solo contiene métodos abstractos (métodos que estudiaremos un poco más adelante).

Las interfaces deben seguir las siguientes reglas:

- Una interfaz no puede contener datos, pero puede contener propiedades, indexadores (consulte la sección Los indexadores, en este capítulo) y eventos.
- Ningún modificador de acceso precede a los elementos de la interfaz; todo es implícitamente de tipo "public".
- Una clase puede implementar tantas interfaces como desee, mientras que solo puede extender de una única clase.
- Una interfaz puede heredar de una o de varias interfaces.

4. Implementación

Imaginemos ahora que el fabricante Electro276 ofrece en su catálogo una tarjeta de entrada/salida llamada E276, cuyas características electrónicas son compatibles con nuestras necesidades. Electro276 libera con esta tarjeta una DLL "clásica" que permite la programación en lenguaje C.

Una DLL (Dynamic Link Library) es un formato de archivo correspondiente a las bibliotecas de software. Concretamente, contiene código ejecutable desde las funciones que se pueden conectar dinámicamente. Ofreciendo una DLL, el constructor de la tarjeta ofrece funciones de "alto nivel" para controlar la tarjeta sin liberar sus secretos de fabricación.

Para utilizar la tarjeta E276 con nuestra aplicación, bastará con construir una clase que implemente la interfaz IBasicIO y que invocará al código de la DLL.

```
class E276: IBasicIO
{

  bool LeerBinario(uint numTarjeta, uint numEntrada)
  {
    //...
    return true;
  }

  int LeerAnalogica(uint numTarjeta, uint numEntrada)
  {
    //...
    return 0;
  }
  void EscribirBinario(uint numTarjeta, uint numSalida, bool estado)
  {
    //...
  }
  void EscribirAnalogica(uint numTarjeta, uint numSalida, int val)
  {
    //...
  }

}
```

Si se declara una clase que hereda de una interfaz pero no implementa todos los miembros de la interfaz, se producirán errores de compilación.

Sintaxis de declaración:

```
class MiClase: IMiInterfaz [, IMiInterfaz2...]
{
   //...
}
```

- IMiInterfaz es la interfaz que se deberá implementar en MiClase.
- IMiInterfaz2, ... es una lista opcional de otras interfaces que MiClase deberá implementar.

5. Visual Studio y las interfaces

Visual Studio nos ayuda en la implementación de las interfaces. Véalo usted mismo:

Declare una interfaz IBasicIO.

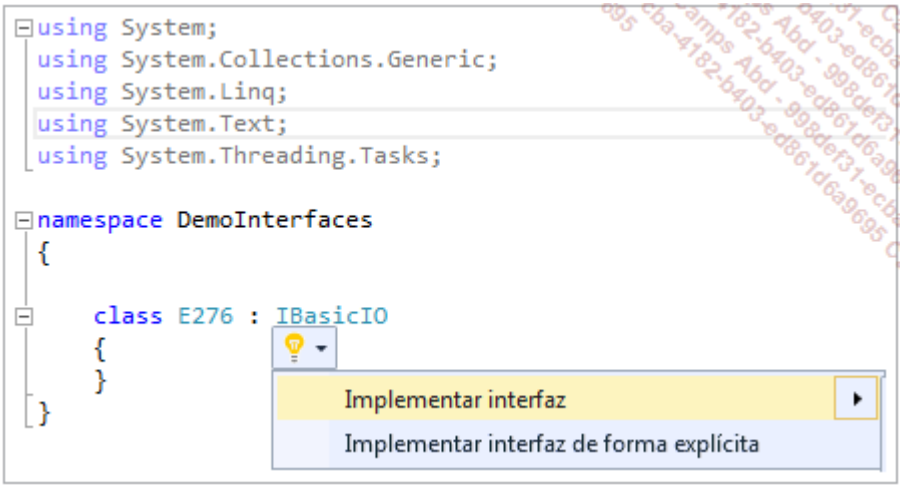
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemoInterfaces
{
  // Interfaz que contiene los métodos a soportar obligatoriamente
  public interface IBasicIO
  {
    // Lectura de una entrada binaria referencia a la tarjeta + número
de entrada
    bool LeerBinario(uint numTarjeta, uint numEntrada);
    // Lectura de una entrada analógica referencia a la tarjeta + número
de entrada
    int LeerAnalogica(uint numTarjeta, uint numEntrada);
```

```
    // Escritura de una salida binaria referencia a la tarjeta + número
de salida
    void EscribirBinario(uint numTarjeta, uint numSalida, bool estado);
    // Escritura de una salida analógica referencia a la tarjeta + número
de salida
    void EscribirAnalogica(uint numTarjeta, uint numSalida, int val);
  }
```

Declare una clase E276 que implemente la interfaz IBasicIO.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemoInterfaces
{

  class E276: IBasicIO
  {
  }
```

Haga clic con el botón derecho del ratón en **IBasicIO** y seleccione **Implementar interfaz**.



Visual Studio crea en la clase E276 todos los métodos de la interfaz IBasicIO. Por tanto, no habrá errores de compilación.

Por el contrario, si deja el código tal cual, se generará una excepción de tipo NotImplementedException durante la ejecución para informarle de que todavía tiene trabajo por hacer.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemoInterfaces
{
  class E276: IBasicIO
  {
    public bool LeerBinario(uint numTarjeta, uint numEntrada)
    {
        throw new NotImplementedException();
    }
    public int LeerAnalogica(uint numTarjeta, uint numEntrada)
    {
        throw new NotImplementedException();
    }
    public void EscribirBinario(
       uint numTarjeta, uint numSalida, bool estado)
    {
        throw new NotImplementedException();
    }
    public void EscribirAnalogica(
        uint numTarjeta, uint numSalida, int val)
    {
        throw new NotImplementedException();
    }
  }
```

6. Representación UML de una interfaz

El diagrama de clases de Visual Studio genera el siguiente gráfico:

Observe la manera que utiliza la implementación (también llamada "realización" en lenguaje UML) de la interfaz para representar la clase E276.

7. Interfaces y polimorfismo

Normalmente, interfaces y polimorfismo van de la mano en la programación orientada a objetos. Gracias al contrato IBasicIO, nuestra aplicación considerará todas las tarjetas de entrada/salida como objetos de tipo IBasicIO. Solo se sabrá en un único sitio que se trata de una tarjeta E276: durante su instanciación.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;


namespace DemoInterfaces
{
  class Program
  {
    static void Main(string[] args)
    {
      IBasicIO miTarjeta = new E276();
      miTarjeta.EscribirAnalogica(0, 5, 0xe5);
    }
  }
}
```

8. Ejercicio

a. Enunciado

Para asentar este concepto de interfaz, vamos a codificar de manera somera una aplicación de consola, que sepa utilizar diferentes medios de comunicación para transferir los datos. En función de la infraestructura disponible, el usuario podrá elegir entre Ethernet, Wifi y 3G.

Cada medio de comunicación se encapsulará en una clase que implementa una interfaz llamada IBaseCom, que contiene los siguientes métodos:

- Conectar
- Escribir
- Leer

- Desconectar

Para simplificar el código, los cuatro métodos no recibirán argumentos y no devolverán nada.

b. Consejos

Creación de la interfaz

Cree un nuevo proyecto de tipo consola llamado TPcomm.

Añada una interfaz llamada IBaseCom que contendrá los cuatro métodos enumerados anteriormente.

Creación de las tres clases de implementación

Se van a implementar tres clases IBaseCom con sus respectivos aspectos específicos.

La implementación de estos cuatro métodos mostrará una línea en la consola, recordando el medio de comunicación y la acción relacionada con el método.

Ejemplo:

3G - Conectar

Crear una clase llamada _3G.

Herede _3G de la interfaz IBaseCom.

Utilice Visual Studio para generar automáticamente los métodos en la clase _3G (clic con el botón derecho del ratón en **IBaseCom - Implementar interfaz**).

Sustituya el código insertado por un comando de visualización en consola, que evoque al medio encapsulado y la acción solicitada.

Ejemplo:

```
System.Console.Out.WriteLine("3G-Conectar");
```

Repita la operación para las clases Wifi y Ethernet.

Codificación de la aplicación

La aplicación debe mostrar un menú que ofrezca los tres medios de comunicación. En función de la elección, se instanciará y memorizará un objeto en forma de tipo IBaseCom. El resto del código llamará a los métodos Conectar, Enviar, Recibir y Desconectar antes de volver a mostrar el menú.

Codifique un bucle de tipo do while.

En este bucle, muestre un menú conciso que presente las tres opciones de comunicación y una opción para salir de la aplicación.

Declare una referencia a un objeto de tipo IBaseCom, con valor null.

Gestione el valor introducido por el usuario y codifique un switch.

En función de la opción seleccionada, instancie la clase asociada y actualice la referencia IBaseCom como salida del switch y, si la referencia no es null, llame a los cuatro métodos.

Ejecución

Una vez que se compila nuestra aplicación, podemos ejecutarla y probar su funcionamiento. En primer lugar se muestra el menú principal. Posteriormente, el usuario elige y valida el tipo de medio que desea utilizar. En función de su elección se traza el encadenamiento de las llamadas en la consola y, posteriormente, la aplicación vuelve a mostrar el menú.

```
Menú prinicipal
1: 3G
2: Wifi
3: Ethernet

0: Salir
Su elección:
1
3G-Conectar
3G-Escribir
3G-Leer
3G-Desconectar

Menú prinicipal
1: 3G
2: Wifi
3: Ethernet
0: Salir
Su elección:
2
Wifi-Conectar
Wifi-Escribir
Wifi-Leer
Wifi-Desconectar

Menú prinicipal
1: 3G
2: Wifi
3: Ethernet

0: Salir
Su elección:
3
Ethernet-Conectar
Ethernet-Escribir
Ethernet-Leer
Ethernet-Desconectar

Menú prinicipal
1: 3G
2: Wifi
3: Ethernet
0: Salir
Su elección:
```

c. Corrección

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TPcomm
{
  interfaz IBaseCom
  {
    void Conectar();
    void Escribir();
    void Leer();
    void Desconectar();
  }

  class _3G: IBaseCom
  {
    public void Conectar()
    {
      System.Console.Out.WriteLine("3G-Conectar");
    }

    public void Escribir()
    {
      System.Console.Out.WriteLine("3G-Escribir");
    }

    public void Leer()
    {
      System.Console.Out.WriteLine("3G-Leer");
    }
```

```
  public void Desconectar()
  {
    System.Console.Out.WriteLine("3G-Desconectar");
  }
}

class Wifi: IBaseCom
{
  public void Conectar()
  {
    System.Console.Out.WriteLine("Wifi-Conectar");
  }

  public void Escribir()
  {
    System.Console.Out.WriteLine("Wifi-Escribir");
  }

  public void Leer()
  {
    System.Console.Out.WriteLine("Wifi-Leer");
  }

  public void Desconectar()
  {
    System.Console.Out.WriteLine("Wifi-Desconectar");
  }
}

class Ethernet: IBaseCom
{
  public void Conectar()
```

```
  {
    System.Console.Out.WriteLine("Ethernet-Conectar");
  }

  public void Escribir()
  {
    System.Console.Out.WriteLine("Ethernet-Escribir");
  }

  public void Leer()
  {
    System.Console.Out.WriteLine("Ethernet-Leer");
  }

  public void Desconectar()
  {
    System.Console.Out.WriteLine("Ethernet-Desconectar");
  }
}



class Program
{
  static void Main(string[] args)
  {
    bool fin = false;
    do
    {
      #region Mostrar el menú
      Console.Out.WriteLine("Menú Principal");
      Console.Out.WriteLine("1: 3G");
      Console.Out.WriteLine("2: Wifi");
      Console.Out.WriteLine("3: Ethernet");
```

```
        Console.Out.WriteLine("");
        Console.Out.WriteLine("0: Salir");
        Console.Out.WriteLine("Su elección: ");
        #endregion

        // Referencia a una instancia de tipo IBaseCom
        IBaseCom baseCom = null;
        #region Gestión elección
        int eleccion = -1;
        if (!int.TryParse(Console.In.ReadLine(), out eleccion))
          eleccion = -1;
        #endregion
        switch (eleccion)
        {
          case 0:
            fin = true;
            break;
          case 1:
            // Instanciación de una clase _3G y
            // almacenamiento de su referencia en baseCom
            // posible porque _3G es un IBaseCom
            baseCom = new _3G();
            break;
          case 2:
            // Idem con la clase Wifi
            baseCom = new Wifi();
            break;
          case 3:
            // Idem con la clase Ethernet
            baseCom = new Ethernet();
            break;
          default:
            Console.Out.WriteLine("ELECCIÓN INVÁLIDA");
```

```
            break;
        }
        if (baseCom != null)
        {   // Si se valida la elección, entonces se asigna baseCom
          baseCom.Conectar();
          baseCom.Escribir();
          baseCom.Leer();
          baseCom.Desconectar();
        }
        Console.Out.WriteLine();
        Console.Out.WriteLine();
      } while (!fin);
    }
  }
}
```

Este ejercicio ha aplicado la implementación de una interfaz por clases y su uso de manera homogénea desde una aplicación.

9. Las interfaces de .NET

.NET ofrece interfaces que permiten a sus objetos integrarse de manera sencilla con el funcionamiento general del framework.

Por ejemplo:

- Ha creado objetos que contienen varias propiedades.
- Almacena estos objetos en tablas (que estudiaremos un poco más adelante).
- Desea poder ordenar estas tablas utilizando el método estándar de .NET: Array.Sort(miTabla).

Problema: solo usted conoce los criterios de clasificación de sus objetos.

Solución: herede su clase de la interfaz "normalizada .NET" IComparable y desarrolle su algoritmo de ordenación en su método CompareTo.

Ejemplo:

A continuación se muestra una clase Vehiculo, que contiene dos propiedades: Constructor y Anio. Para poder clasificar varios objetos de este tipo en una

tabla, la clase Vehiculo implementa la interfaz IComparable y expone su método-contrato CompareTo. En este método está la lógica de comparación entre dos objetos.

```
  // La clase Vehiculo hereda de IComparable y
  // por tanto debe implementar el método CompareTo
  class Vehiculo: IComparable
  {
    public String Constructor { get; set; }
    public int Anio { get; set; }

    // El método CompareTo contiene nuestra lógica
    // de comparación entre instancias de Vehiculo
    public int CompareTo(object obj)
    {
      Vehiculo v = obj as Vehiculo;
      return String.Compare(this.Constructor, v.Constructor);
    }
  }
```

Todavía no hemos visto las tablas, pero sepa desde ahora que gracias a la implementación de la interfaz IComparable se podrá ordenar una colección de varias instancias de nuestra clase Vehiculo utilizando métodos de alto nivel como Array.Sort.

A continuación se muestra un ejemplo de código donde descubrimos la sintaxis de construcción de una tabla, cómo se rellena con objetos de nuestra clase Vehiculo y la iteración con sus elementos, antes y después de la ordenación.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemoIComparableEIComparar
{
  class Program
  {
    static void Main(string[] args)
```

```
  {
    Vehiculo[] tabVehiculos = new Vehiculo[]
    {
      new Vehiculo(){ Constructor = "Renault"},
      new Vehiculo(){ Constructor = "Citroën"},
      new Vehiculo(){ Constructor = "Peugeot"}
    };

    Console.WriteLine("Contenido no ordenado:");
    foreach (var item in tabVehiculos)
    {
      Console.WriteLine("\t" + item.Constructor);
    }

    // Solicita la ordenación mediante el método static Array.Sort
    Array.Sort(tabVehiculos);

    Console.WriteLine("Contenido ordenado:");
    foreach (var item in tabVehiculos)
    {
      Console.WriteLine("\t" + item.Constructor);
    }
  }
}

// La clase Vehiculo hereda de IComparable y
// por lo tanto debe implementar el método CompareTo
class Vehiculo: IComparable
{
  public String Constructor { get; set; }
  public int Anio { get; set; }

  // El método CompareTo contiene nuestra lógica
```

```
    // de comparación entre instancias de Vehiculo
    public int CompareTo(object obj)
    {
      Vehiculo v = obj as Vehiculo;
      return String.Compare(this.Constructor, v.Constructor);
    }
  }

}
```

La ejecución de este código provoca la siguiente visualización por consola:

```
Contenido no ordenado:
        Renault
        Citroën
        Peugeot
Contenido ordenado:
        Citroën
        Peugeot
        Renault
Pulse una tecla para continuar...
```

Asociación, composición y agregación

En todo programa, el desarrollador tiene que diseñar clases que utilizan o contienen otras clases que, a su vez, pueden utilizar o contener otras clases. Por ejemplo, un formulario (cuadro de diálogo con el usuario) muestra diferentes controles, como botones de selección, casillas de selección, campos de introducción de texto o listas desplegables. El formulario y cada uno de sus controles se "encapsulan" en clases que el desarrollador va a asociar para conseguir el formulario final.

Las asociaciones son más o menos fuertes. En nuestro ejemplo, la asociación es fuerte porque es el formulario el que instancia estos controles. Estos mismos controles se destruirán cuando se cierre. Hablamos de agregación "compuesta".

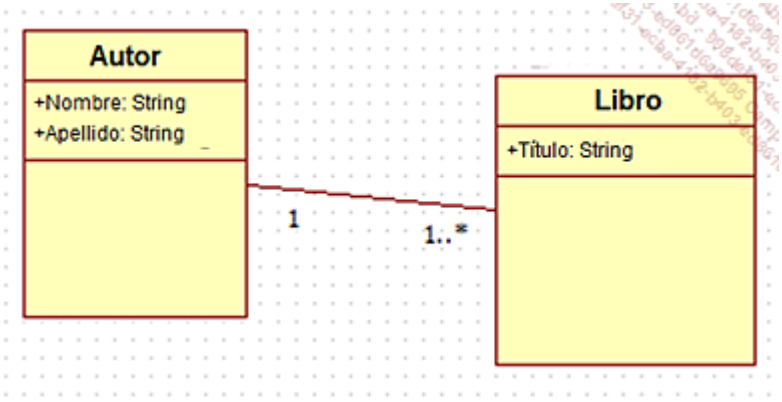
Mientras dura esta asociación, el objeto continente accede libremente a los miembros de tipo public de cada uno de los objetos contenidos. De esta manera, durante su carga, nuestro formulario podrá inicializar los contenidos por defecto de las cajas de texto, las selecciones de

los botones de selección y, durante la validación, recuperar las selecciones del usuario, preguntando a cada uno de los controles.

¿Cómo C# permite gestionar estas diferentes formas de colaboración?

Independientemente de cuál sea el grado de asociación, la clase host necesitará almacenar referencias a las clases contenidas.

Puede tener varios objetos del mismo tipo referenciado en la clase host. Esta pluralidad se expresa en UML con un índice en el extremo de la relación indicando una cantidad fija o un rango posible.



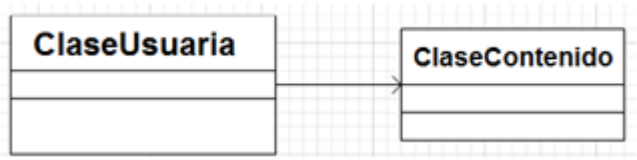
En este ejemplo, un autor ha escrito uno o un número indefinido de libros y un libro pertenece a un solo autor.

C# va a tener varias formas de codificación para estos diferentes tipos de asociación.

- La clase "continente" tiene una sencilla referencia a un objeto de tipo "contenido".

```
class Usuaria
{
   Contenido contenido  = null;
}
```

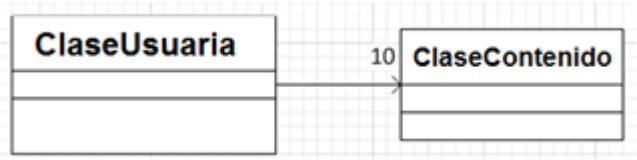

Traducción UML de esta asociación:



- La clase "continente" tiene una lista de referencias de tamaño fijo a los tipos de "contenido". En este caso, será preferible un objeto de tipo tabla, como se verá más adelante.

```
class Continente
{
   Contenido[] tabContenidos = new Contenido[10];
}
```

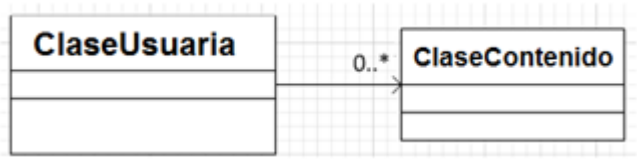
Traducción UML de esta asociación:



- La clase "continente" tiene una lista de tamaño indefinido de referencias a tipos "contenido". En este caso, es preferible un objeto del tipo List<>, como se verá más adelante.

```
class Continente
{
   List<Contenido> listaContenidos = new List<Contenido>;
}
```

Traducción UML de esta asociación:



Volveremos sobre las tablas y las colecciones genéricas.

Observe que en estos tres ejemplos se declaran las ubicaciones para almacenar las referencias, aunque los objetos de tipo Contenido todavía no se han instanciado.

En el caso de una asociación "sencilla", la clase Usuaria no instancia los objetos a los que referencia. Los recibirá "del exterior", por ejemplo, como argumentos en uno de sus métodos.

```
class Continente

{

   Contenido contenido = null;

   public SetContenido(Contenido contenido)

   {

      this.contenido = contenido;

   }

}
```

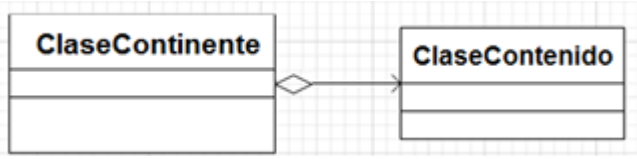
En este caso, el objeto Contenido puede sobrevivir a la clase Continente (salvo si nadie la referencia).

En el caso de una asociación "fuerte", el objeto Continente se encarga de crear el o los objetos Contenido. El momento de esta creación, es decisión del desarrollador. Lo que importa es que el objeto se creará antes de su uso.

Primera sintaxis posible:

```
class Continente

{

   Contenido contenido = new Contenido();

}
```

Traducción UML de esta asociación:



El objeto contenido se crea con el objeto continente. Es una manera radical de tratar el problema de la no disponibilidad, pero el constructor del objeto contenido solo podrá recibir información conocida durante la ejecución.

Se trata de una forma de agregación por referencia (rombo vacío en la figura anterior), porque el elemento contenido se instancia por el elemento continente.

Recordemos que en el caso de una composición (rombo relleno en la figura anterior), el "contenido" forma parte del "continente"; solo puede pertenecer al "continente" y se destruye al mismo tiempo que este último. La composición o agregación por valor de objetos de tipo Referencia, no es posible en C#. Al contrario que en C++, es obligatorio que se instancie una clase en el heap y, por tanto, continente y contenido siempre tendrán zonas de memoria distintas. Para que el contenido se fusione con el continente es necesario que pertenezca a la familia de los valores. El tipo estructura, que veremos más adelante, es muy parecido a la clase y forma parte de la familia de los valores.

Si desea forzar lo más posible una composición con objetos de tipo Referencia debe codificarla a partir de una agregación por valor y tener cuidado con nunca transmitir al exterior las referencias de los objetos contenidos. De esta manera, el ciclo de vida de los objetos contenidos se corresponderá con el de los objetos continentes.

```
using System;


namespace Cap5
{
  class Program
  {
    static void Main(string[] args)
    {
      new Prueba();
    }


    public class Contenido
    {
      public void MetodoContenido()
      {
        //...
      }
    }


    public class Continente
    {
```

```
      private Contenido contenido = new Contenido();


      public void MetodoContinente()
      {
        contenido.MetodoContenido();
        //...
      }
    }
  }


  class Prueba
  {
    public Prueba()
    {
      Program.Continente continente
        = new Program.Continente();
      continente.MetodoContinente();
    }
  }


}
```

En este fragmento de código, el miembro contenido se crea al mismo tiempo que el objeto continente. El objeto contenido no se puede copiar al exterior, porque ningún método del objeto continente devuelve su referencia y su atributo de visibilidad prohíbe el acceso directo. Por tanto, se seleccionará durante la destrucción tan pronto como el continente se destruya.

Volvamos a la sintaxis de declaración/instanciación de un objeto "continente" en un objeto "contenido". Esta forma de creación tiene el defecto de no poder recibir información del runtime. Si esta restricción es un problema, basta con mover la instanciación en uno de los métodos de la clase "continente", como su constructor, que puede ser una alternativa interesante.

```
class Continente

{

   Contenido contenido = null;

   public Continente(string info)

   {

      this.contenido = new Contenido(info);

    }

}
```

En el ejemplo, podemos imaginar que el constructor de la clase "contenido" espera una cadena de caracteres dinámica, como una entrada del usuario. En este caso, el encadenamiento es ideal, pero el nuevo defecto de esta solución es que el objeto contenido se crea sistemáticamente. Puede que necesitemos sus servicios en determinados casos.

La siguiente solución solo crea el objeto cuando es necesario.

```
class Continente
{
  Contenido contenido = null;
  private string info;
  public Continente(string info)
  {
    this.info = info;
  }

  public void Tratamiento()
  {
    //...
    this.contenido = new Contenido(this.info);
    //...
  }

  public void DespuesDeTratamiento()
    {
       //...
       this.contenido.Accion():
       //...
    }
}
```

Si a través de un método el objeto "continente" devuelve una referencia de su objeto "contenido" a una instancia superior que la almacena y la explota, entonces los objetos "contenido" y "continente" serán independientes. Por ejemplo, el objeto "contenido" podrá sobrevivir a la destrucción del objeto continente que lo haya creado.

Si no queda registrada ninguna referencia al objeto "contenido" por otros objetos, "continente" y "contenido" desaparecerán juntos...

Ahora es momento de volver a las tablas y su codificación en C#.

1. Las tablas

Las tablas son sucesiones de referencias o valores dispuestos de manera contigua. Una tabla contiene elementos de tipos iguales. Las tablas tienen un tamaño fijo, declarado en su creación. Modificar una tabla para añadir una nueva entrada durante una operación no es muy sencillo; hay que crear una segunda tabla con el nuevo tamaño y copiar en ella los datos de la anterior y añadir la nueva entrada. Si en un programa el tamaño de la colección puede cambiar durante la ejecución de manera frecuente por operaciones de inserción o borrado, es mejor utilizar colecciones, que se presentarán en el capítulo Herencia y polimorfismo.

Sin embargo, la tabla, encapsulada por la clase System.Array, es el medio más básico para contener series de datos.

La declaración de la tabla indica el tipo de datos que va a contener y el número de entradas que va a soportar.

Sintaxis de creación de una tabla:

```
tipo[] nombreTabla = new tipo [tamaño];
```

Ejemplo:

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

En este ejemplo, tabInt contiene diez enteros "listos para usar", porque el tipo int forma parte de la familia Valor y no es necesaria otra forma de instanciación.

```
Contenido[] tabContenidos = new Contenido[10];
```

En este segundo ejemplo, tabContenidos está lista para recibir diez referencias de tipo Contenido (que es una clase), pero atención: estas referencias todavía no están definidas.

Cada entrada de la tabla es accesible por un índice numérico, que va de 0 al tamaño de la tabla menos uno. En nuestro ejemplo, será posible acceder desde tabContenidos[0] hasta tabContenidos[9].

Cualquier tabla hereda de la clase System.Array, por tanto forma parte de la familia de las referencias, independientemente del tipo de datos que contenga. Hereda una importante serie de métodos y propiedades.

A continuación se muestra una selección de los más utilizados:

Clear()	Método de tipo static que permite reinicializar todas las entradas de una tabla.
CopyTo()	Realiza una copia de los elementos a una segunda tabla.
Length	Propiedad que devuelve el número de entradas de una tabla.
Rank	Propiedad que devuelve el número de dimensiones de la tabla.
Reverse()	Invierte el contenido de una tabla en una dimensión.
Sort()	Método de tipo static que permite realizar la ordenación de una tabla en una dimensión. Si los objetos de la tabla heredan de la interfaz IComparable, su método CompareTo se utiliza directamente para realizar la clasificación.

La inicialización de una tabla se puede realizar directamente en la declaración de la clase:

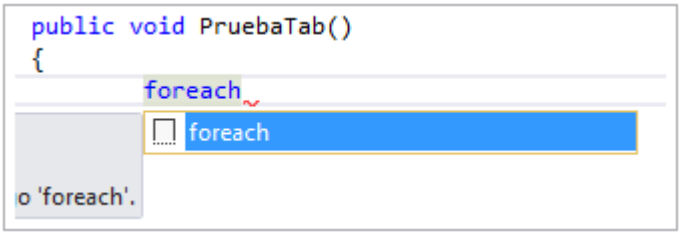
```
class MiClase
{
        string[] miTablaCadenas = new string[]
               { "Cadena1", "Cadena2", "Cadena3" };
        //...
}
```

En este fragmento de código no se precisa el tamaño de la tabla porque el compilador la puede deducir durante la definición de las entradas.

La inicialización también se puede hacer en un método, como el constructor de la clase, por ejemplo:

```
class MiClase
{

     string[] miTablaCadenas = null;

     public MiClase()
     {
          miTablaCadenas = new string[3];
          miTablaCadenas[0] = "Cadena1";
          miTablaCadenas[1] = "Cadena2";
          miTablaCadenas[2] = "Cadena3";
     }
 }
```

Recorrer el contenido de una tabla se puede hacer sencillamente con la sintaxis foreach, que el asistente de Visual Studio le ayuda a escribir. Para ello, escriba la palabra clave foreach.



Después pulse dos veces la tecla de tabulación.

```
public void PruebaTab()
{
  foreach (var item in collection)
  {

  }
}
```

Visual Studio ha insertado automáticamente el código de la iteración de una colección. El uso de la palabra clave var evita declarar el tipo leído, porque se puede deducir por el compilador en función de la declaración de la tabla. En cada vuelta de este bucle, la variable item contendrá una entrada de la tabla.

```
    public void PruebaTab()
    {
        foreach (var item in miTablaCadenas)
        {
            Console.WriteLine(item);
        }
    }
```

Un bucle foreach permite un único recorrido previo.

También puede utilizarse un bucle for para recorrer una tabla. La lectura de cada entrada se hará por el operador []. Visual Studio le ayuda a escribirla si utiliza el mismo procedimiento que se ha explicado con anterioridad, es decir, escribir for y después pulsar dos veces la tecla de tabulación.

```
    public void PruebaTab2()
    {
        for (int i = 0; i < length; i++)
        {


        }
    }
```

La sintaxis for contiene tres partes separadas por ";"

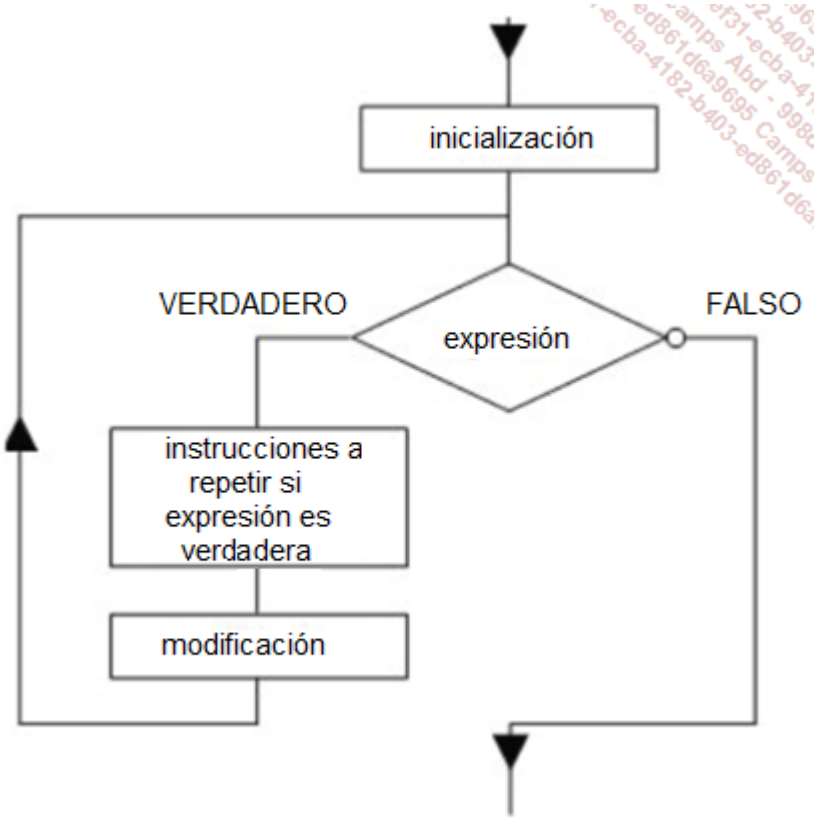
```
for(<inicialización>;<expresión_condición>;<modificación>)
{}
```

La primera de las tres partes fija a 0 el valor de inicio, es decir, una variable local al bucle de tipo int, llamada i.

La segunda representa una comprobación que se realizará al inicio de cada paso del bucle, incluida la primera. Aquí, i se compara con una variable simbólica llamada length que hay que sustituir por el tamaño de nuestra tabla: miTabCadenas.Length. Si la comprobación se cumple, se ejecutará el contenido del bucle. En caso contrario, la ejecución pasará al siguiente paso.

La última parte contiene la operación que se realizará al final del bucle, aquí el incremento de la variable local i.

Representación algorítmica del bucle for:




```
public void PruebaTab2()
{
    for (int i = 0; i < miTablaCadenas.Length; i++)
    {
        Console.WriteLine(miTablaCadenas[i]);
    }
}
```

Durante la ejecución de la iteración, la variable i va a variar de 0 hasta "tamaño de la tabla -1", porque la condición es mientras que i sea más pequeño que el tamaño de la tabla. Por tanto, la variable i se puede utilizar como índice de acceso a la tabla.

La sintaxis es:

```
<nombreTabla>[índice]
```

El contenido de las celdas de una tabla puede modificarse durante la ejecución de la aplicación.

```
    public void PruebaTab3(int índice, string nuevo)
    {
        miTablaCadenas[índice] = nuevo;
    }
```

Por el contrario, es imposible añadir o borrar elementos de la tabla directamente.

Una tabla se puede pasar como argumento a un método y también ser devuelto por un método.

El ciclo de vida de una tabla se corresponde con el ciclo de vida de un objeto de tipo Referencia: si nadie más la utiliza, se convierte en destino para el garbage collector. Si la tabla contiene tipos valor, desaparecerán con él. Si contiene tipos Referencia, sus entradas podrán ser destruidas si nadie más las referencia. Podemos imaginar aquí la complejidad de la operación del garbage collector.

Una tabla puede tener varias dimensiones. El siguiente ejemplo instancia una tabla de dos dimensiones. Cada dimensión tiene 10 entradas.

```
int[,] tabPyt = new int[10, 10];
```

El siguiente código permite rellenar las entradas de esta tabla de dos dimensiones realizando el producto de los índices.

```
    public void PruebaTab4()
    {
        for (int i = 0; i < 10; i++)
        {
            for (int j= 0; j < 10; j++)
            {
                tabPyt[i, j] = i * j;
            }
        }
    }
```

También es posible utilizar una tabla de tablas.

Declaración de una tabla de tablas:

```
<tipo>[][] nombreTabla = new <tipo>[<tamaño>][];
```

Ejemplo:

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

Después, hay que asignar una tabla para cada entrada de la tabla. Las tablas asignadas pueden ser de tamaños diferentes.

```
    int[][] miTablaDeTablas = new int[10][];
    for (int i = 0; i < miTablaDeTablas.Length; i++)
    {
        miTablaDeTablas[i] = new int[5];
        //...
    }
```

Para acceder a los elementos de la tabla hay que utilizar la siguiente sintaxis:

```
nombreTabla[<índice tabla principal>][<índice tabla secundaria>]
```

Ejemplo:

```
    miTablaDeTablas[2][3] = 7;
```

2. Las colecciones

Hemos visto que si el programa debe poder insertar, eliminar o añadir elementos a una lista, el uso de colecciones es obligatorio.

El espacio de nombres System.Collections.Generic contiene una serie de clases que permiten gestionar las colecciones. La sintaxis de declaración de las colecciones "genéricas" externaliza el tipo de los objetos a almacenar como argumento. Este tipo de objeto se define entre dos ángulos, precedidos por el tipo de la colección.

Hay colecciones "no genéricas" y, por tanto, débilmente tipadas. Estas colecciones "no genéricas" fueron las primeras versiones de las colecciones de .NET y su uso ahora se desaconseja.

Sintaxis de una declaración de colección:

```
List<miTipo> nombreColección = new List< miTipo >();
Con miTipo para el tipo de elemento de la colección
```

Ejemplo:

```
    List<string> miColecCadenas = new List<string>();
```

Este fragmento de código instancia una colección de string (fuertemente tipada).

Las clases que gestionan las colecciones generalmente se representan con la letra T, que indica el tipo a almacenar, por ejemplo, List<T>. Son convenciones de escritura: T para "tipo", TKey para "clave" y TValue para "valor".

El uso de colecciones genéricas simplifica el código y optimiza la ejecución, porque no hay ninguna conversión que realizar en modo lectura o en modo escritura de los datos. En todos los métodos de la colección, el argumento T se sustituye por el tipo almacenado.

Por ejemplo, la clase List<T> implementa un método public void Add(T item);.

Si en una clase Continente se declara una colección de tipo List<T> de objetos Contenido...

```
class Continente

{

   List<Contenido> listaContenidos = new List<Contenido>;

   //...

}
```

... entonces el método public void Add(T item); se transforma para la instancia listaContenidos en void Add(Contenido item);.

Este principio que consiste en situar como argumento el tipo utilizado para una entidad de operación se utiliza mucho en .NET. En la declaración encontramos clases, pero también en las interfaces, estructuras y como argumento de determinados métodos.

Volvamos a las colecciones genéricas y veamos las principales clases que se proponen.

a. List<> y LinkedList<>

La clase List<T> es una colección muy próxima al tipo tabla. Se utiliza cuando el programa no hace muchas inserciones pero la rapidez de acceso a las celdas es importante. El almacenamiento utilizado internamente por la clase es de tipo tabla, con celdas contiguas, autorizando así el acceso directo a un índice particular.

El siguiente fragmento de código muestra la instanciación de un objeto de tipo List de int. El método Add permite añadir directamente valores a la lista. En la clase List<T>, que soporta la interfaz IEnumerable, es posible realizar una iteración usando la palabra clave foreach. El contenido de la lista se modifica dinámicamente antes de ser recorrida de nuevo, esta vez por un bucle for.

```
List<int> lista = new List<int>();
lista.Add(400);
lista.Add(5);
lista.Add(28);

foreach (var item in lista)
{
    Console.WriteLine(item);
}
Console.WriteLine("");

lista.Add(300);
lista.RemoveAt(1);
for (int i = 0; i < lista.Count; i++)
{
    Console.WriteLine(lista[i]);
}
```

Salida por la consola:

```
400
5
28

400
28
300
```

Como se muestra en el siguiente fragmento de código, se puede añadir a un objeto de tipo List<T> el contenido de una tabla "clásica", y el objeto List<T> puede devolver una tabla (type System.Array).

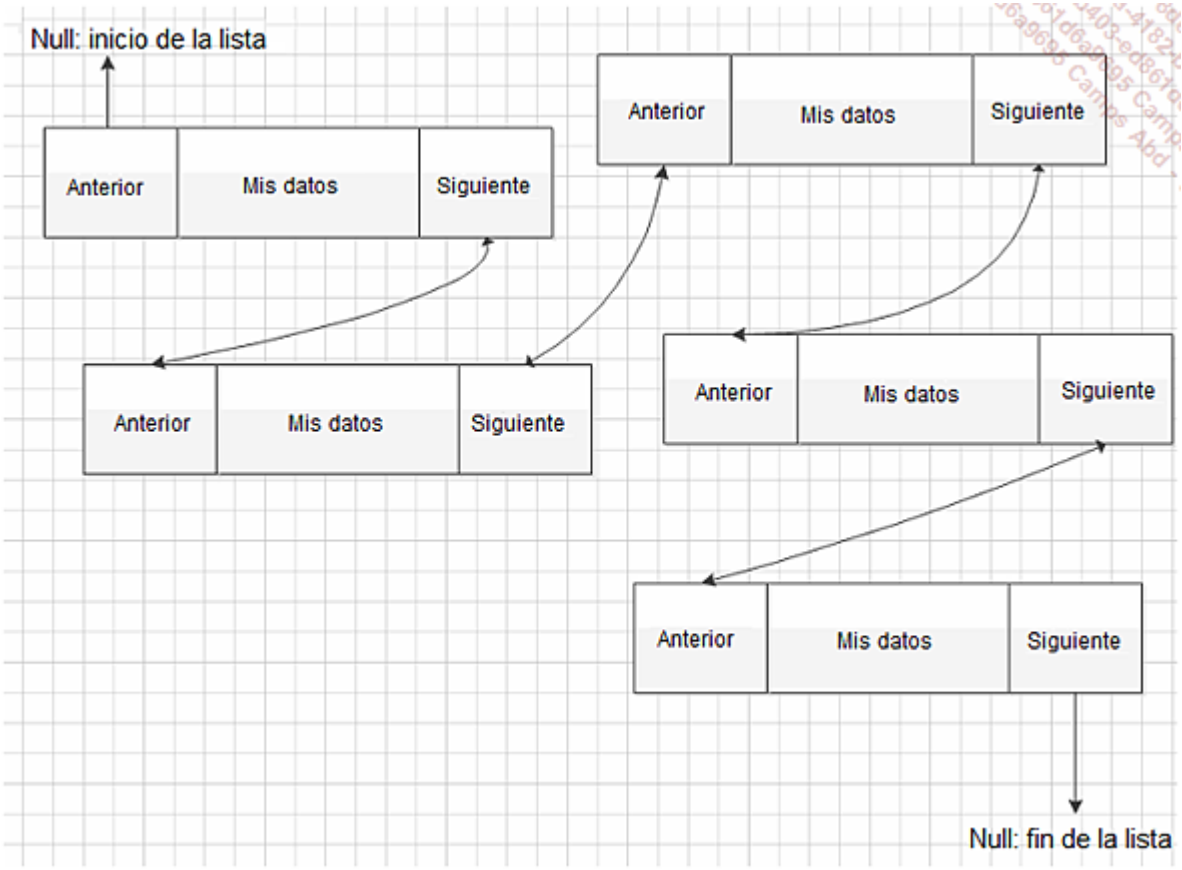
```
List<int> lista = new List<int>();
lista.Add(400);
lista.Add(5);
lista.Add(28);

int[] entrada = new int[] { 1, 2, 3, 4, 5 };
lista.AddRange(entrada);

int[] salida = lista.ToArray();
foreach (var item in salida)
{
    Console.WriteLine(item);
}
```

Salida por la consola:

```
400
5
28
1
2
3
4
5
```

Hay otros métodos y propiedades implementados en la clase List<T>. La documentación en línea de Microsoft, disponible en el sitio MSDN (Microsoft Developer Network) y accesible desde la URL http://msdn.microsoft.com, aporta a los desarrolladores una descripción de las clases y ejemplos de uso.

La clase LinkedList<T> es una colección basada en una lista encadenada. En una lista encadenada cada celda contiene, además del elemento almacenado (referencia o valor), las referencias a las celdas anterior y siguiente.



Con una asociación como esta, inserciones, cambios de orden y eliminación, son muy rápidos porque se trata de conectar con los miembros Siguiente y Anterior. Por el contrario, el acceso directo a un índice concreto no está soportado. Para simular este funcionamiento de tipo "tabla" hay que recorrer la colección desde su inicio, contando cada celda hasta alcanzar el índice de la posición solicitada.

b. Queue<T> y Stack<T>

La clase Queue<T> es una colección diseñada para un funcionamiento de tipo FIFO (first in - first out). Por ejemplo, este tipo de colecciones se utiliza cuando se debe memorizar temporalmente información, porque su operación no se puede realizar inmediatamente. Durante la operación de inicio, el orden de lectura se debe corresponder con el orden de entrada en la fila, de ahí la denominación FIFO "primero en entrar, primero en salir". Tendremos la ocasión de volver sobre este tipo de objetos.

La clase Stack<T> es una colección diseñada para un funcionamiento de tipo LIFO (last in - first out), que es el comportamiento inverso del modelo FIFO.

c. Dictionary<TKey, TValue>

Un diccionario es una colección de valores a los que se puede acceder rápidamente usando una clave. Por ejemplo, si queremos conocer rápidamente el número de cualquier mes del año, se podrá construir un diccionario que tenga como claves los nombres de los meses y como valores números enteros de 1 a 12. La clase Dictionary<TKey, TValue> se adapta a este tipo de uso.

```
// Instanciación del diccionario:
Dictionary<string, int> losMeses = new Dictionary<string, int>();


// Llenar las parejas clave - valor
losMeses.Add("Enero", 1);
losMeses.Add("Febrero", 2);
losMeses.Add("Marzo", 3);
losMeses.Add("Abril", 4);
losMeses.Add("Mayo", 5);
losMeses.Add("Junio", 6);
losMeses.Add("Julio", 7);
losMeses.Add("Agosto", 8);
losMeses.Add("Septiembre", 9);
losMeses.Add("Octubre", 10);
losMeses.Add("Noviembre", 11);
losMeses.Add("Diciembre", 12);


// Utilización del diccionario:


Console.WriteLine("El número del mes de junio es "+losMeses["Junio"]);
```

d. Los enumeradores

En programación orientada a objetos siempre se busca la abstracción. Puede ser interesante poder cambiar de tipo de colección, por ejemplo, pasar de una List<T> a una LinkedList<T> sin tener que volver a escribir el código que hace las iteraciones.

Las colecciones que implementan la interfaz IEnumerable (como List, LinkedList Queue, Stack y Dictionary), permiten devolver un objeto que permite realizar sus enumeraciones. Estos objetos se llaman enumeradores. Implementan la interfaz IEnumerator<T> y se pueden pasar como argumento a los métodos, como en el siguiente fragmento de código:

```
void MuestraColeccion(IEnumerator<string> en)
{
    while (en.MoveNext())
    {
        string s = en.Current;
        Console.WriteLine(s);
    }
}
```

A continuación se muestra un fragmento de código que rellena diferentes tipos de colecciones y que utiliza el método genérico definido anteriormente para mostrar su contenido.

```
Queue<string> q = new Queue<string>();
q.Enqueue("aaa");
q.Enqueue("bbb");
q.Enqueue("ccc");
MuestraColeccion(q.GetEnumerator());
List<string> l = new List<string>();
l.Add("ddd");
l.Add("eee");
l.Add("fff");
MuestraColeccion(l.GetEnumerator());

LinkedList<string> ll = new LinkedList<string>();
ll.AddFirst("ggg");
ll.AddFirst("hhh");
ll.AddFirst("iii");
MuestraColeccion(ll.GetEnumerator());

Stack<string> st = new Stack<string>();
st.Push("jjj");
st.Push("kkk");
st.Push("lll");
MuestraColeccion(st.GetEnumerator());
```

El método MuestraColeccion realmente ha hecho abstracción del tipo de la colección fuente.

e. La magia del yield

Imagine que tiene un objeto que almacena una gran colección y que los usuarios de este objeto piden esta colección con diferentes filtros. Siempre puede construir colecciones temporales, fruto de las operaciones de filtrado solicitadas, y devolverlas a los métodos adecuados. O también, puede utilizar la palabra clave yield que le evitará estas listas intermedias.

Esta palabra clave mágica, una vez más, se utiliza dentro de una iteración. Utilizada con un return permite devolver el contenido de una celda conservando el estado de la iteración actual para que la siguiente llamada pueda volver a empezar donde se quedó. Veamos todo esto con un ejemplo concreto.

```
class DemoYield
{
    int[] minNumeros = new int[] { 1, 2, 3, 4, 5, 6, 7, 8 };


    public IEnumerable<int> GetNumerosPares()
    {
        foreach (var item in minNumeros)
        {
            if (item % 2 == 0)
                yield return item;
        }
    }
    public IEnumerable<int> GetNumerosImpares()
    {
        foreach (var item in minNumeros)
        {
            if (item % 2 != 0)
                yield return item;
        }
    }
}


static void Main(string[] args)
{
```

```
    DemoYield dy = new Program.DemoYield();
    Console.WriteLine("GetNumerosPares");
    foreach (var item in dy.GetNumerosPares())
    {
        Console.WriteLine(item.ToString());
    }
    Console.WriteLine("GetNumerosImpares");
    foreach (var item in dy.GetNumerosImpares())
    {
        Console.WriteLine(item.ToString());
    }
}
```

La clase DemoYield contiene una colección de enteros y dos métodos para extraer los valores pares e impares en forma de listas IEnumerable<int>. Estos dos métodos realizan una iteración foreach que aplica un filtrado y devuelve los valores correspondientes gracias a:

```
yield return item;
```

La clase que usa los servicios de DemoYield ve sus dos métodos como colecciones.

```
foreach (var item in dy.GetNumerosPares())
```

No se ha construido ninguna lista intermedia y, una vez más, C# simplifica nuestra codificación.

La palabra clave yield se utiliza mucho en las enumeraciones construidas por consultas LINQ.

La secuencia yield break permite detener «antes de tiempo» la ejecución de la iteración. De hecho, si durante la iteración vemos que es inútil continuar más adelante, durante la ejecución de yield break se informará al código que invoca de que ha terminado.

3. Ejercicio

a. Enunciado

Crear una nueva solución de tipo consola.

En el Main, crear una tabla que contenga los días de la semana.

Muestre cada día utilizando un bucle foreach.

Crear una colección de tipo ArrayList.

Copiar de nuevo el contenido en orden inverso de la tabla anterior en la colección.

Mostrar la colección.

Eliminar de la colección las entradas que contengan "Martes" y "Jueves" (utilizar LastIndexOf y RemoveAt).

Mostrar la colección.

Crear una colección fuertemente tipada string (System.Collections.Generic.List<string>).

Copiar de nuevo el contenido de la tabla en esta colección de string.

Mostrar la colección.

Crear un diccionario con las claves de tipo string y los valores de tipo int (System.Collections.Generic.Dictionary<string, int>).

Copiar de nuevo el contenido de la tabla de los días en este diccionario, pasando como valores los números de los días (1 para el Lunes, etc.).

Utilizar el método ContainsKey para comprobar la presencia de Miércoles y mostrar su valor, utilizando la sintaxis ["Miércoles"] en el diccionario.

Salida por consola asociada:

```
Domingo
Sábado
Viernes
Jueves
Miércoles
Martes
Lunes

Domingo
Sábado
Viernes
Miércoles
```

```
Lunes


Lunes
Martes
Miércoles
Jueves
Viernes
Sábado
Domingo


Miércoles es el día número 3 de la semana
```

b. Corrección

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LabCollections
{
  class Program
  {
      static void Main(string[] args)
      {
        string[] tab
          = new string[]{
            "Lunes", "Martes", "Miércoles",
            "Jueves", "Viernes", "Sábado",
            "Domingo"};

        foreach (var item in tab)
        {
          System.Console.WriteLine(item);
```

```
        }
        System.Console.Write("\n");



        System.Collections.ArrayList coleccion
          = new System.Collections.ArrayList();
        for (int i = tab.Length - 1; i >= 0; i--)
        {
          coleccion.Add(tab[i]);
        }

        foreach (var item in coleccion)
        {
          System.Console.WriteLine(item as string);
        }
        System.Console.Write("\n");

        int k = coleccion.LastIndexOf("Martes");
        if (k != -1)
          coleccion.RemoveAt(k);
        k = coleccion.LastIndexOf("Jueves");
        if (k != -1)
          coleccion.RemoveAt(k);

        foreach (var item in coleccion)
        {
          System.Console.WriteLine(item as string);
        }
        System.Console.Write("\n");

        System.Collections.Generic.List<string> coleccionTipo
          = new List<string>();
        foreach (var item in tab)
```

```
        {
          coleccionTipo.Add(item);
        }
        foreach (var item in coleccionTipo)
        {
          System.Console.WriteLine(item );
        }

        Dictionary<string, int> dicoDays
          = new Dictionary<string, int>();
        for (int i = 0; i < tab.Length; i++)
        {
          dicoDays.Add(tab[i], i + 1);
        }
        if (dicoDays.ContainsKey("Miércoles"))
        {
          System.Console.Write("\n");
          System.Console.WriteLine(
            "Miércoles es el día número {0} de la semana",
            dicoDays["Miércoles"]);
        }
        System.Console.ReadKey();
    }
  }
}
```

Las clases anidadas

Es posible declarar una clase en una clase. Esta funcionalidad ofrece al desarrollador otra manera de organizar su código. Las clases principales se registran en los espacio de nombres; así es posible realizar una nueva clasificación dentro de estas clases principales.

La mayor parte del tiempo, la clase anidada, llamada nested class o inner class, no significa nada fuera de su clase host y su operador de visibilidad es de tipo private. A pesar de todo, es posible modificar este tipo de acceso a public o protected.

Sintaxis de una nested class:

```
class Host
{
   class Anidada
   {
   }
}
```

La clase anidada tiene acceso a todos los miembros de la clase host. La clase host solo tiene acceso a los miembros de tipo public de la clase anidada.

El hecho de declarar una clase en otra es una forma de notación. No tiene ninguna consecuencia inducida en las instanciaciones de las clases host y anidada, que siguen siendo independientes.

Ejemplo de codificación de una clase anidada y su clase host:

```
using System;
namespace Cap7
{
  class Program
  {
    static void Main(string[] args)
    {
      new Prueba();
    }
    class ClaseHost
    {
      class ClaseAnidada
      {
        public ClaseHost MiHost { get; set; }
```

```
        public string PropPublicaClaseAnidada { get; set; }
        private string PropPrivadaClaseAnidada { get; set; }

        public void PruebaClaseHost(ClaseHost ch)
        {
          this.MiHost = ch;
          Console.WriteLine(this.MiHost.PropPrivadaClaseHost);
          Console.WriteLine(this.MiHost.PropPublicaClaseHost);
        }
      }
      public string PropPublicaClaseHost { get; set; }
      private string PropPrivadaClaseHost { get; set; }
      public void PruebaHostAnidado()
      {
        this.PropPrivadaClaseHost = "Hello";
        this.PropPublicaClaseHost = "World";
        ClaseAnidada cn = new ClaseAnidada();
        cn.PruebaClaseHost(this);
        Console.WriteLine(cn.PropPublicaClaseAnidada);
      }
    }
    class Prueba
    {
      public Prueba()
      {
        ClaseHost ch = new ClaseHost();
        ch.PruebaHostAnidado();
      }
    }
  }
}
```

Salida correspondiente por la consola:

```
Hello
World


Pulse una tecla para continuar...
```

Este fragmento de código muestra una clase host, ClaseHost, que instancia una clase anidada, ClaseAnidada, durante la llamada a su método PruebaHostAnidado. La instancia de esta clase anidada accede a los miembros de su host, de tipo public o private.

Si la clase anidada se declara de tipo public, se convertirá en utilizable desde "el exterior". Sencillamente habría que indicar su ruta de acceso durante su instanciación.

```
using System;


namespace Cap7
{
  class Program
  {
    static void Main(string[] args)
    {
      new Prueba();
    }


    public class ClaseHost
    {
      public class ClaseAnidada
      {
        public string PropPublicaClaseAnidada { get; set; }
      }


      public string PropPublicaClaseHost { get; set; }
    }
  }


  class Prueba
  {
```

```
    public Prueba()
    {
      Program.ClaseHost.ClaseAnidada cn
        = new Program.ClaseHost.ClaseAnidada();
      cn.PropPublicaClaseAnidada = "Hello";
    }
  }
}
```

Las estructuras

Las estructuras son una herencia de los lenguajes C y C++. En sus formas C# se parecen mucho a las clases, pero con una diferencia de tamaño: las estructuras se asignan a la pila, a diferencia de las clases, que se asignan a la cola. Esto tiene dos consecuencias:

- El ciclo de vida de la estructura se limita a las llaves que la encierran.
- La estructura no necesita gestión de la memoria (asignación, garbage collector), de modo que tiene un rendimiento muy alto para entidades muy pequeñas.

Las estructuras no existen en Java.

1. Declaración de una estructura

Sintaxis de declaración de una estructura:

```
[visibilidad] struct nombre [:interfaces] {//implementación } [;]
```

El atributo de visibilidad de una estructura definida en un espacio de nombres, puede ser de tipo:

- public: la estructura es utilizable por todos.
- internal: la estructura es utilizable por los componentes del ensamblado que lo alberga.

Los atributos private, protected y protected internal solo tienen sentido para las estructuras anidadas. Una estructura anidada es una estructura definida en una clase.

Precedida del atributo private, la estructura anidada solo se podrá utilizar en su clase host.

Precedida del atributo protected, la estructura anidada solo se podrá utilizar por las clases heredadas de su clase host.

Precedida del atributo protected internal, la estructura anidada solo se podrá utilizar por las clases heredadas de su clase host en el ensamblado.

El nombre de una estructura respeta las mismas reglas que los nombres de las clases.

Una estructura puede heredar de una o varias interfaces. Se indican después del nombre de la estructura. Cada uno se separada de la siguiente por una coma. Las reglas de implementación son las mismas que para una clase, a saber, tienen que implementar la lista de los miembros definidos en la interfaz.

Una estructura no puede heredar de otra estructura o de una clase diferente de Object. No puede servir de "base" a una clase.

Una estructura C#, igual que todo en .NET, hereda de System.Object.

Como para una clase, el cuerpo de la estructura se sitúa entre llaves.

El punto y coma final es opcional.

Los tipos sencillos ofrecidos por C# como bool, double o int son estructuras, como se comprueba en el siguiente fragmento.

```
Assembly mscorlib.dll, v4.0.0.0


using System.Globalization;

using System.Runtime;

using System.Runtime.InteropServices;

using System.Security;


namespace System

{

  ... public struct Int32: IComparable, IFormattable,

IConvertible, IComparable<int>, IEquatable<int>

  {

    ... public const int MaxValue = 2147483647;

    ... public const int MinValue = -2147483648;
```

La estructura, al formar parte de la familia de los valores, se comporta de modo que el operador = realiza una copia de los miembros de tipo datos y no una copia de referencia.

Cuando una estructura se pasa como argumento de tipo Valor a un método, la CLR realiza una copia en la llamada.

Una estructura solo puede declarar constructores con argumento(s).

Una estructura no puede declarar el destructor.

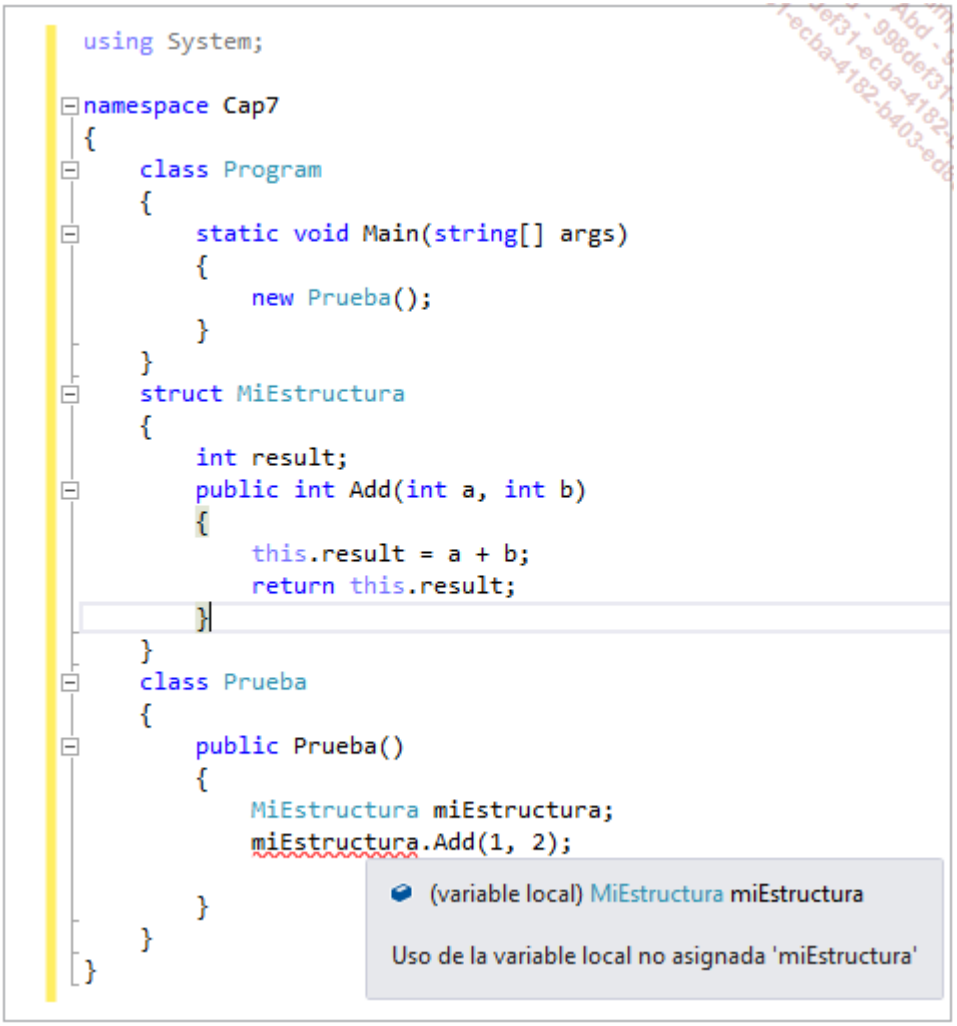
2. Instanciación de una estructura

Como el tipo estructura forma parte de la familia de los valores, se puede instanciar directamente como sigue:

```
using System;


namespace Cap7
{
  class Program
  {
    static void Main(string[] args)
    {
      new Prueba();
    }
  }
  struct MiEstructura
  {
    public int Add(int a, int b)
    {
      return a + b;
    }
  }
  class Prueba
  {
    public Prueba()
    {
      MiEstructura miEstructura;
      miEstructura.Add(1, 2);
    }
  }
}
```

En este caso, el compilador no falla con la estructura, que solo contiene un método y ningún atributo.

Por el contrario, si se añade a la estructura un miembro variable, por ejemplo de tipo int, el compilador devuelve un error sobre el uso de una estructura no inicializada. Su dato no se ha inicializado.



Para corregir el problema basta con instanciar la estructura con la palabra clave new como sigue:

```
MiEstructura miEstructura = new MiEstructura();
```

Con esta sintaxis, cada miembro de tipo variable se inicializa por una llamada a su constructor. En nuestro caso, sabemos que el tipo int no es otro que un alias del tipo System.Int32 y que el constructor de este objeto inicializa su valor a 0.

Si el valor por defecto fijado por los constructores no conviene, siempre es posible inicializar uno a uno los miembros variables de la estructura, con la condición de que dispongan de un atributo de visibilidad adecuado.

```
using System;


namespace Cap5
{
  class Program
  {
    static void Main(string[] args)
    {
      new Prueba();
    }
  }
  struct MiEstructura
  {
    public int resultado;
    public int Add(int a, int b)
    {
      this.resultado = a + b;
      return this.resultado;
    }
  }
  class Prueba
  {
    public Prueba()
    {
      MiEstructura miEstructura;
      miEstructura.resultado = 300;
      miEstructura.Add(1, 2);
    }
  }
}
```

Las clases parciales

La mayor parte de la veces, una clase se define en un único archivo fuente (de extensión .cs). Si varios desarrolladores deben enriquecer la misma clase al mismo tiempo, la implementación en un archivo único plantea problemas. De la misma manera, Visual Studio ofrece un asistente gráfico que permite diseñar cuadros de diálogo y generar código C# que los encapsula en una clase. Esta clase se "comparte" con el desarrollador. Por ejemplo, Visual Studio escribe todo el código que permite instanciar y situar un botón en el cuadro de diálogo; también prepara el marco del método que se invoca cuando el usuario hace clic en este botón. Le corresponde al desarrollador "rellenar" el cuerpo del método. Compartir el mismo archivo fuente entre el diseñador gráfico y el desarrollador puede ser complicado y C# ofrece una solución elegante que autoriza el fraccionamiento de clases (y también de estructuras y de interfaces) en varios archivos. Durante la compilación, el código se fusiona y la clase se reconstituye.

Se utiliza la palabra clave partial para indicar al compilador que la clase se puede completar por otros archivos fuente.

Ejemplo de contenido del primer archivo fuente MiClaseCompartida.cs

```
namespace Cap5
{
  partial class MiClaseCompartida
  {
    public int MiPropiedad01 { get; set; }
  }
}
```

Ejemplo de contenido del segundo archivo fuente MiClaseCompartida2.cs

```
namespace Cap5
{
  partial class MiClaseCompartida
  {
    public int MiPropiedad02 { get; set; }
  }
}
```

Se definen algunas reglas para este modo de funcionamiento:

- La clase fraccionada se debe definir cada vez en el mismo espacio de nombres y la palabra clave partial siempre debe preceder a sus definiciones.
- Tan pronto como un módulo declara una herencia, se convierte en activo para el resto y esto sin tener que declararlo en el resto.
- Tan pronto como un módulo declara la clase cerrada (sealed) o abstracta (abstract), la propiedad se activa para el resto.
- Los miembros declarados en un módulo son accesibles en el resto.
- Los atributos de compilación se fusionan.

Los métodos parciales

Cuando la definición de una clase se fracciona en varios archivos fuente, es posible definir la firma de métodos en un lado para eventualmente implementarlos en otro. Es esta noción de eventualidad la que tiene todo el interés de esta funcionalidad. Si el compilador encuentra una implementación del método en uno los archivos fuente, la integra. En caso contrario, retira la definición del método.

Varias reglas para este modo de codificación:

- Los métodos de tipo partial siempre deben tener void como tipo de retorno.
- No definir ni atributo de visibilidad ni modificador a los métodos de tipo partial. Siempre son implícitamente de tipo private.

Ejemplo de contenido del primer archivo fuente MiClaseCompartida.cs:

```
namespace Cap5
{
  partial class MiClaseCompartida
  {
    partial void MiMetodoParcial(string param1);
  }
}
```

Ejemplo de contenido del segundo archivo fuente MiClaseCompartida2.cs

```
namespace Cap5
{
  partial class MiClaseCompartida: Prueba
  {
    partial void MiMetodoParcial(string param1)
    {
      System.Console.WriteLine(param1);
    }
  }
}
```

Los indexadores

C# permite construir tipos que ofrecen acceso a sus colecciones internas utilizando el operador de índice ([ ]) como si se tratara de tablas nativas.

Imaginemos por ejemplo una clase Empleados que gestiona una lista de empleados. La clase define un determinado número de métodos para calcular el tiempo de permanencia, los salarios, etc., y contiene una colección de objetos que describen cada empleado. Esta colección no será de tipo public en virtud de la regla de encapsulación, pero el usuario de la clase deberá poder acceder a cualquier objeto Empleado de la manera más sencilla posible...

Proponer una sintaxis basada en el operador [ ] para acceder a una ubicación particular de la colección es una solución muy acertada.

Ejemplo:

```
Empleado e = MisEmpleados[3];
```

La implementación de la sobrecarga del operador [ ] se realiza como sigue:

```
visibilidad tipoRetorno this[TipoIndice valorIndice]
{
   // descriptores de acceso get y set
}
```

- La mayor parte de las veces, el atributo de visibilidad será de tipo public.
- El tipoRetorno corresponde al tipo contenido en la colección.
- De nuevo se utiliza this para simbolizar la instancia.
- [TipoIndice valorIndice] define el tipo y el valor a pasar para identificar el índice al que acceder en la colección. En el caso de una colección de tipo List<T>, se utilizará un int. En el caso de una colección "diccionario", el tipo podrá ser cualquier otro.

- La sintaxis de los descriptores de acceso set y get es la misma que la utilizada en la codificación de las propiedades.

A continuación se muestra un fragmento de código que presenta un ejemplo de codificación de un indexador:

```
using System.Collections.Generic;


namespace DemoIndexador
{
  class Program
  {
    static void Main(string[] args)
    {
      Empleados empleados = new Empleados();
      empleados[0] = new Empleado() { Nombre = "Ángel" };
      empleados[1] = new Empleado() { Nombre = "José" };
      empleados[2] = new Empleado() { Nombre = "María" };
    }
  }

  class Empleado
  {
    public string Nombre { get; set; }
  }

  class Empleados
  {
    private List<Empleado> coleccion
      = new List<Empleado>();

    // ... métodos varios

    public Empleado this[int indice]
    {
      set
```

```
      {
        this.coleccion.Insert(indice,valor);
      }
      get
      {
        return this.coleccion[indice];
      }
    }
  }
}
```

Los indexadores se pueden implementar en las clases, las estructuras y las interfaces.

Ejemplo de interfaz con indexador:

```
  interfaz IGetEmpleado
  {
    Empleado this[int indice] { set; get; }
  }
```

Es importante que el código del descriptor de acceso asegure el acceso a la colección comprobando que el índice que se pasa está incluido en sus límites y que el tipo del objeto que se pasa como argumento durante una escritura es el esperado.

El siguiente código comprueba la coherencia del índice y el tipo recibido en modo escritura en la colección. Utiliza el mecanismo de las excepciones para devolver el error al código que lo invoca. Por tanto, el usuario debería utilizar una estructura try catch para la llamada.

```
using System;
using System.Collections.Generic;


namespace DemoIndexador
{
  class Program
  {
    static void Main(string[] args)
    {
```

```
    try
    {
      Empleados empleados = new Empleados();
      empleados[0] = new Empleado() { Nombre = "Ángel" };
      empleados[1] = new Empleado() { Nombre = "José" };
      empleados[2] = new Empleado() { Nombre = "María" };
      empleados[3] = null;


      Empleado e = empleados[3];

    }
    catch (IndexOutOfRangeException)
    {
      Console.WriteLine("Índice fuera del límite");
    }
    catch (ArgumentoExcepcion)
    {
      Console.WriteLine("Tipo incompatible");
    }
  }
}

class Empleado
{
  public string Nombre { get; set; }
}



interfaz IGetEmpleado
{
  Empleado this[int indice] { set; get; }
}
```

```
  class Empleados: IGetEmpleado
  {
    private List<Empleado> coleccion
      = new List<Empleado>();
    // ... métodos varios

    public Empleado this[int indice]
    {
      set
      {
        if (!(valor is Empleado))
          throw new ArgumentoExcepcion();

        if (indice >= 0 && indice <= this.coleccion.Count)
          this.coleccion.Insert(indice,valor);
        else
          throw new IndexOutOfRangeException();
      }
      get
      {
        if (indice >= 0 && indice < this.coleccion.Count)
          return this.coleccion[indice];
        else
          throw new IndexOutOfRangeException();
      }
    }
  }
}
```

Sobrecarga de operadores

Acabamos de ver que el indexador con el operador [ ] era un medio práctico que se podía ofrecer a los usuarios de las clases para acceder a las entradas de sus colecciones.

Es posible sobrecargar otros operadores si esto tiene sentido para sus objetos.

Tomemos ejemplo del operador +. Se usa entre dos enteros y permite realizar su suma. El operador + se "sobrecargó" en la clase System.String para realizar la concatenación de dos cadenas.

Otro ejemplo: el operador ==; por defecto, si los tipos a comparar forman parte de la familia Valor, serán valores que se podrán comparar. En caso contrario lo serán las referencias. En el caso de System.String, que forma parte de la familia referencia, será el contenido de las cadenas lo que se comparará, porque el operador == se sobrecargó para esto.

Sobrecargar los operadores siempre genera controversia entre los desarrolladores. De hecho hay que "navegar" en la documentación de la clase utilizada para saber cómo se comportan los operadores. Además, Java ha rechazado completamente esta funcionalidad.

Sobrecargar el operador ==

Como ya hemos visto, System.Object implementa un método virtual Equals. Este método se puede sobrecargar en nuestras clases para que la comparación, se haga, por defecto, sobre las referencias de las clases, y después se realiza sobre los valores "pertinentes" contenidos en las clases.

Un uso incluso más intuitivo de la clase consistiría en utilizar el operador == para comprobar la igualdad de los contenidos. Para ello hay que sobrecargarlo.

Sintaxis de sobrecarga del operador que comprueba de igualdad:

```
public static bool operador ==(Tipo nombreInstancia1, Tipo
nombreInstancia2){ ... }
```

En C# la sobrecarga del operador que comprueba de igualdad obligatoriamente debe estar acompañada por la del operador de desigualdad:

```
public static bool operador !=(Tipo nombreInstancia1, Tipo
nombreInstancia2){ ... }
```

Como siempre hay que evitar la redundancia de código, el método Equals generalmente es central y contiene el cuerpo de instrucciones que realizan la comprobación de igualdad. Los operadores == y != solo serán dos puntos de acceso adicionales.

Para garantizar coherencia en las comprobaciones de igualdad, la sobrecarga de los operadores == y != se acompaña par la sobrecarga de los

métodos System.Object.Equals y System.Object.GetHashCode. Esto no es una obligación sino una buena práctica.

Recordemos que el método GetHashCode, también método virtual de System.Object, devuelve un int, que es el identificador de la instancia del objeto. Si sobrecarga el método Equals, tendrá que sobrecargar GetHashCode porque estos dos métodos se llaman juntos por diferentes componentes de .NET.

Por defecto, el método GetHashCode se basa en la dirección de memoria del objeto para construir el valor entero de identificación. Cuando se quieren integrar los estados del objeto en lugar de su dirección de memoria en la comprobación de igualdad será necesario utilizar los datos de la instancia y no los datos de tipo static. Por ejemplo, si la clase representa un cliente de la empresa, su hashcode podrá devolver su número de cliente porque es único.

Ejemplo de código completo:

```
using System;


namespace DemoSobrecargaOperador
{
  class Program
  {
    static void Main(string[] args)
    {
      MiClase prueba1 = new MiClase()
      { MiInt = 1, MiCadena = "Hello" };
      MiClase prueba2 = new MiClase()
      { MiInt = 1, MiCadena = "Hello" };
      if (prueba1 == prueba2)
        Console.WriteLine(
          "prueba1 y prueba2 tienen contenidos idénticos");
      Console.WriteLine(
        "Hashcode de prueba1: " + prueba1.GetHashCode());
      Console.WriteLine(
        "Hashcode de prueba2: " + prueba2.GetHashCode());
    }
  }
  class MiClase
  {
```

```
    public int MiInt { get; set; }
    public string MiCadena { get; set; }

    public override bool Equals(object obj)
    {
      if (obj is MiClase)
      {
        MiClase mc = obj as MiClase;
        return this.MiInt == mc.MiInt
          && this.MiCadena == mc.MiCadena;
      }
      return false;
    }

    public override int GetHashCode()
    {
      return MiInt ^ MiCadena.GetHashCode();
    }

    static public bool operador ==(MiClase mc1, MiClase mc2)
    {
      return mc1.Equals(mc2);
    }

    static public bool operador !=(MiClase mc1, MiClase mc2)
    {
      return !(mc1 == mc2);
    }
  }
}
```

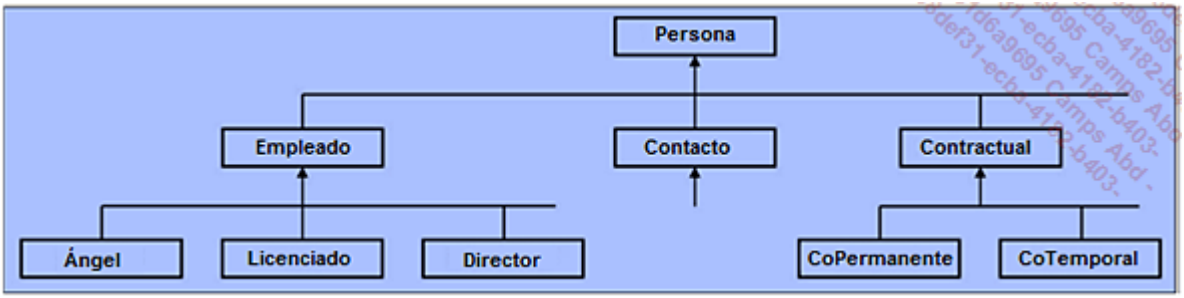
Entender la herencia

El mecanismo de la herencia se utiliza mucho en programación orientada a objetos y es importante recordar su utilidad.

Heredar una clase influye en la "especialización" de determinados comportamientos y propiedades, aprovechando sus servicios básicos y, de esta manera, evitar redundancia de código.

C# solo permite una herencia por nivel (a diferencia de C++), aunque es posible heredar de una clase la cual sea heredada y así sucesivamente para construir una herencia de clases que parta de la más global hasta la más detallada.

Ejemplo de herencia de clases:



Una clase puede servir de "madre" para varias clases. El mejor ejemplo es, sin ninguna duda, System.Object, que es raíz de todos los tipos y, por tanto, de todas las clases de C#. Recordemos que la herencia de System.Object es implícita y no necesita ninguna definición particular.

Codificación de la clase de base y su heredada

El código de una clase define las reglas de su eventual herencia.

1. Prohibir la herencia

En primer lugar, ¿es deseable hacer una clase "heredable"? Si el análisis demuestra que no, entonces se debe utilizar la palabra clave sealed (cerrada) en la definición de la clase para prohibir cualquier herencia.

Sintaxis de declaración de una clase cerrada:

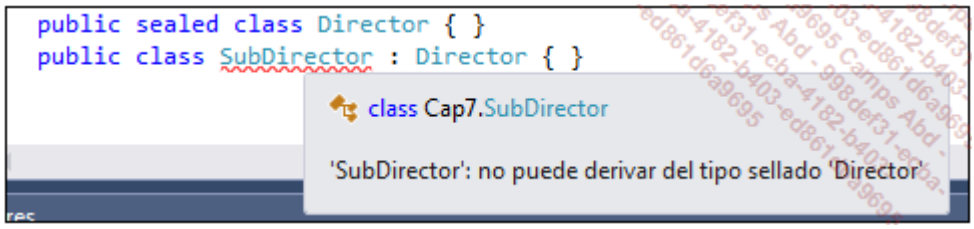
```
[visibilidad] sealed class NombreClase
```

```
{
   //...
}
```

Ejemplo de clase cerrada:

```
    public sealed class Director
    {
        //...
    }
```

Como se muestra en la siguiente captura de imagen, el compilador rechaza una clase que hereda de Director:



La clase .NET System.String es una clase cerrada.

Una clase que no contiene la palabra clave sealed en su definición se considera como heredable.

2. Definir miembros heredables

Una clase de base define sus miembros "heredables" gracias a sus atributos de accesibilidad. De esta manera, las clases heredadas tendrán permiso de utilizar y redefinir los miembros de tipo protected y los de tipo public. Los miembros de tipo private seguirán siendo no accesibles.

3. Codificación de la herencia

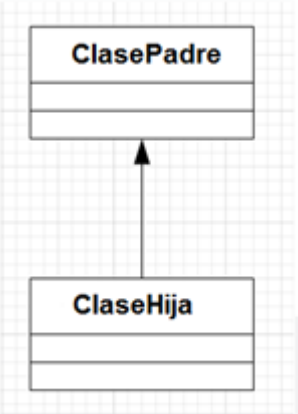
Sintaxis de declaración de la herencia de una clase:

```
[visibilidad] [sealed] class NombreClaseHeredera: NombreClaseDeBase
{
   //...
}
```

Ejemplo de clase heredada:

```
    class ClaseHija: ClasePadre
    {
        //...
    }
```

Representación UML:



4. Explotación de una clase heredada

La instanciación de un objeto de tipo ClaseHija se hará de manera "clásica" mediante la llamada a new ClaseHija();.

Los miembros accesibles por la referencia del objeto serán los miembros de tipo public de ClaseHija, así como los miembros de tipo public de ClasePadre.

Se utiliza la notación punteada para acceder a los miembros deseados.

Ejemplo de código que muestra una herencia y su uso desde un programa:

```
using System;
```

```
namespace Cap6
{
    class Program
    {
        static void Main(string[] args)
        {
            new Prueba();
        }
        class ClasePadre
        {
            public string PropClasePadre { get; set; }
        }

        class ClaseHija: ClasePadre
        {
            public string PropClaseHija { get; set; }
        }

        class Prueba
        {
            public Prueba()
            {
                ClaseHija dev = new ClaseHija();
                dev.PropClaseHija = "Hija";
                dev.PropClasePadre = ”Padre”;
            }
        }
    }
}
```

El fragmento de código anterior muestra la instanciación de un objeto de tipo ClaseHija y después el acceso a los miembros de tipo public que están en la clase de base y en la clase derivada.

Comunicación entre clase de base y clase heredada

1. Los constructores

Cuando se instancia una clase heredada, se llama al constructor de su clase de base, antes que al suyo. A continuación se muestra un fragmento de código seguido del resultado en la consola, que lo atestigua.

```
using System;


namespace Cap6
{
    class Program
    {
        static void Main(string[] args)
        {
            new Prueba();
        }
        class ClasePadre
        {
            public ClasePadre()
            {
                Console.WriteLine("Ctor ClasePadre");
            }
            public string PropClasePadre { get; set; }
        }


        class ClaseHija: ClasePadre
        {
            public ClaseHija()
            {
                Console.WriteLine("Ctor ClaseHija");
            }
```

```
            public string PropClaseHija { get; set; }


        }
        class Prueba
        {
            public Prueba()
            {
                Console.WriteLine("Creación objeto ClaseHija");
                ClaseHija dev = new ClaseHija();
            }
        }
    }
}
```

Salida correspondiente por la consola:

```
Creación objeto ClaseHija
Ctor ClasePadre
Ctor ClaseHija
Pulse una tecla para continuar...
```

Parece muy claro que este "encadenamiento" ha tenido lugar sin ninguna llamada particular al código de la clase heredada. En el caso de los constructores sobrecargados, este automatismo se puede modificar, porque el encadenamiento se realiza en el constructor adhoc de la clase de base. Gracias a la palabra clave base, la clase heredada va a poder modificar la conexión.

La palabra clave base sirve para acceder a los miembros de la clase de base a partir de una clase derivada.

Sintaxis de llamada de un constructor básico desde un constructor de clase heredada:

```
ClaseHija([tipo argumento1, ...]): base([argumento...])
{
```

```
}
```

Ejemplo de código que realiza el encadenamiento con la palabra clave base:

```
using System;

namespace Cap6
{
    class Program
    {
        static void Main(string[] args)
        {
            new Prueba();
        }
        class ClasePadre
        {
            public ClasePadre()
            {
                Console.WriteLine("Ctor ClasePadre");
            }
            public ClasePadre(string propClasePadre)
            {
                PropClasePadre = propClasePadre;
                Console.WriteLine(
                    "Ctor ClasePadre con argumento");
            }
            public string PropClasePadre { get; set; }
        }

        class ClaseHija: ClasePadre
        {
            public ClaseHija()
            {
                Console.WriteLine("Ctor ClaseHija");
            }
```

```
            public ClaseHija(string propClasePadre,
                             string propClaseHija)
               : base(propClasePadre)
            {
                PropClaseHija = propClaseHija;
                Console.WriteLine(
                    "Ctor ClaseHija con argumentos");
            }
            public string PropClaseHija { get; set; }
        }

        class Prueba
        {
            public Prueba()
            {
                Console.WriteLine("Creación objeto ClaseHija");
                ClaseHija dev =
                    new ClaseHija("Hello", "World");
            }
        }
    }
}
```

Salida por la consola asociada:

```
Creación objeto ClaseHija
Ctor ClasePadre con argumento
Ctor ClaseHija con argumentos
Pulse una tecla para continuar...
```

La modificación del encadenamiento de los constructores se realiza gracias a la línea:

```
: base(propClasePadre)
```

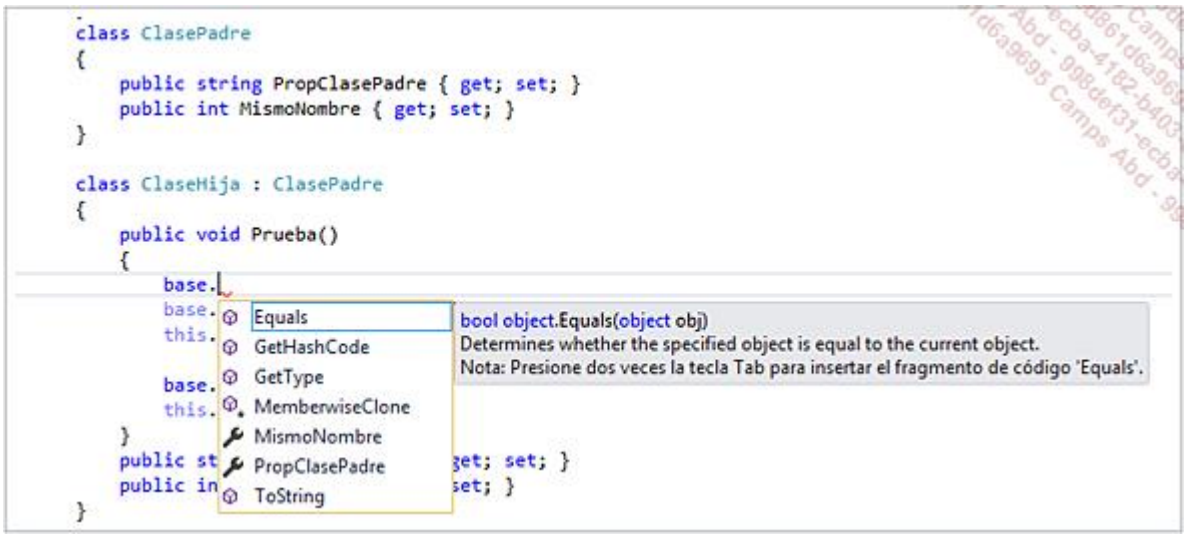
La palabra clave base solo tiene sentido para las instancias de clases. Utilizar base en un método estático provoca un error de compilación.

En Java, el equivalente de la palabra clave base es super.

En C++, hay que enunciar el nombre de la clase madre seguida de ::, para eliminar la ambigüedad de una herencia múltiple

2. Acceso a los miembros básicos desde la clase heredada

También se usa la palabra clave base para que un método de la clase heredada pueda acceder a los miembros de su clase de base. Por supuesto, si el miembro a obtener tiene un nombre único, entonces el uso de la palabra clave base es opcional. Por una parte, el uso de la palabra clave base tiene la ventaja de mantener un código comprensible además de poder activar IntelliSense para que ofrezca todos los miembros heredados accesibles, como demuestra la siguiente captura de pantalla.



Evidentemente, si la clase heredada ofrece un miembro con el mismo nombre y el mismo tipo, la palabra clave base es obligatoria para evitar la ambigüedad.

```
using System;
```

```
namespace Cap6
{
    class Program
    {
        static void Main(string[] args)
        {
            new Prueba();
        }
        class ClasePadre
        {
            public string PropClasePadre { get; set; }
            public int MismoNombre { get; set; }
        }

        class ClaseHija: ClasePadre
        {
            public void Prueba()
            {
                base.PropClasePadre = "Hello";
                this.PropClaseHija = "World";

                base.MismoNombre = 0;
                this.MismoNombre = 1;
            }
            public string PropClaseHija { get; set; }
            public int MismoNombre { get; set; }
        }

        class Prueba
        {
            public Prueba()
            {
```

```
                ClaseHija dev = new ClaseHija();
                dev.Prueba();
            }
        }
    }
}
```

En este fragmento de código, el método Prueba de ClaseHija mezcla el uso básico y this para acceder a los miembros heredados para el primero y a los miembros de la instancia para el segundo. Esta sintaxis solo es obligatoria para la propiedad MismoNombre, que se define en las dos clases. Este caso puede parecer sorprendente en un primer momento. De hecho, ¿por qué nombrar los miembros de manera idéntica? Sin embargo, es una práctica muy corriente y estudiaremos su interés en el capítulo que trata del polimorfismo.

3. Ocultación o substitución de miembros heredados

"Especializando" una clase de base el desarrollador añade nuevos miembros, pero también puede sustituir determinados miembros existentes de la clase de base para conseguir comportamientos específicos.

Se puede ocultar o sustituir métodos, propiedades o indexadores.

Esta codificación se puede realizar de dos maneras: la clase heredada "oculta" los miembros heredados o los "sustituye".

La diferencia de comportamiento resultante se relaciona con el tipo de referencia que va a recibir la instanciación del objeto heredado.

Recordemos que en programación orientada a objetos una clase heredada se puede considerar como un tipo del tipo de la clase de base. Por ejemplo, si una clase Tecnico hereda de la clase Empleado, entonces el objeto de tipo Tecnico se podrá considerar como un objeto de tipo Empleado. Por tanto, su instanciación también se podrá registrar tanto como una referencia de tipo Tecnico como una referencia del tipo Empleado.

El siguiente fragmento de código utiliza este mecanismo.

```
using System;
```

```
namespace Cap6
{
    class Program
    {
        static void Main(string[] args)
        {
            new Prueba();
        }
        class ClasePadre
        {
            public string PropClasePadre { get; set; }
        }

        class ClaseHija: ClasePadre
        {
            public string PropClaseHija { get; set; }
        }
        class Prueba
        {
            public Prueba()
            {
                ClaseHija dev = new ClaseHija();
                dev.PropClaseHija = "Hello";

                ClasePadre dev2 = new ClaseHija();
                (dev2 as ClaseHija).PropClaseHija
                    = "Hello";
            }
        }
    }
}
```

Recordemos que una referencia (dev y dev2 en el ejemplo) no es otra cosa que "la dirección" del objeto real. Disponer en memoria de un objeto heredado comienza por la

parte "objeto de base" y a continuación la parte "miembros añadidos". Por tanto es comprensible que la dirección de inicio de esta zona se pueda considerar como referencia del tipo de base o como referencia del tipo clase heredada.

Observe en el ejemplo que cuando el objeto se referencia por su tipo de base es necesario utilizar el operador as TipoHeredero para poder acceder a sus miembros.

Como hemos visto con anterioridad, una clase heredada puede tener miembros de definiciones que se parezcan a los de su clase de base.

En el marco de una substitución, el acceso a los miembros de la clase de base se refiere al miembro redefinido en la clase heredada y esto ocurre sea cual sea la referencia utilizada: base o clase heredada.

La ocultación es más sencilla de entender. Una clase heredada "oculta" un miembro de su clase de base redefiniéndolo (mismo nombre y mismo tipo para los atributos, incluso firma para los métodos) en su código. El enrutamiento descrito anteriormente no está activo y el miembro redefinido se utiliza desde una referencia en la clase heredada únicamente.

A continuación se muestra una tabla que resume los dos modos y el resultado obtenido durante la llamada de un miembro definido al mismo tiempo en la clase de base y en la clase heredada.

Llamada al miembro desde una referencia de	Con ocultación	Con sustitución
tipo clase de base	Miembro básico llamado	Miembro heredado llamado
tipo clase heredada	Miembro heredado llamado	Miembro heredado llamado

a. Codificación de la ocultación

Del lado de la clase heredada, la palabra clave new (equivalente a nuevo) es la que se va a utilizar para indicar al compilador que el miembro del mismo nombre y del mismo tipo que el de la clase de base no es un error. Si la palabra clave new se omite, el compilador devolverá una advertencia.

Del lado de la clase de base, el miembro necesita ser accesible.

A continuación se muestra un ejemplo de código de ocultación y del uso desde los dos tipos de referencia:

```
using System;

namespace Cap6
{
    class Program
    {
        static void Main(string[] args)
        {
            new Prueba();
        }

        class ClasePadre
        {
            public virtual void Accion ()
            {
                Console.WriteLine("ClasePadre.Accion()");
            }
        }

        class ClaseHijaConNew: ClasePadre
        {
            public new virtual void Accion()
            {
                Console.WriteLine(
                    "ClaseHijaConNew.Accion()");
            }
        }

        class Prueba
        {
            public Prueba()
            {
                ClaseHijaConNew dev
```

```
                    = new ClaseHijaConNew();
                dev.Accion();


                ClasePadre dev2
                    = new ClaseHijaConNew();
                dev2.Accion();
            }
        }
    }
}
```

El método miembro Accion de la clase ClasePadre se oculta cuando se llama desde una referencia a la clase ClaseHijaConNew. Sigue estando activa cuando se llamada desde una referencia a ClasePadre.

A continuación se muestra la salida por la consola asociada:

```
ClaseHijaConNew.Accion ()
ClasePadre.Accion()
Pulse una tecla para continuar...
```

b. Codificación de la sustitución

La clase de base debe declarar el miembro con la palabra clave virtual para autorizar su substitución (override) en una clase heredada.

La clase heredada debe utilizar la palabra clave override para indicar al compilador que este miembro va a sustituir a su homónimo en la clase de base.

A continuación se muestra un ejemplo de código de substitución y un uso desde los dos tipos de referencia:

```
using System;

namespace Cap6
{
    class Program
    {
        static void Main(string[] args)
        {
            new Prueba();
        }

        class ClasePadre
        {
            public virtual void Accion()
            {
                Console.WriteLine("ClasePadre.Accion ()");
            }
        }

        class ClaseHijaConOverride: ClasePadre
        {
            public override void Accion()
            {
                Console.WriteLine(
                    "ClaseHijaConOverride.Accion ()");
            }
        }



        class Prueba
        {
            public Prueba()
            {
```

```
                ClaseHijaConOverride dev
                    = new ClaseHijaConOverride();
                dev.Accion();

                ClasePadre dev2
                    = new ClaseHijaConOverride();
                dev.Accion();
            }
        }
    }
}
ClaseHijaConOverride.Accion()
ClaseHijaConOverride.Accion()
Pulse una tecla para continuar...
```

La palabra clave virtual no implica necesariamente una substitución del miembro en las clases heredadas.

Ejercicio

1. Enunciado

Crear una nueva solución de tipo Consola.

Añada una clase CuentaBancaria, que representa una cuenta bancaria con las siguientes propiedades:

- Titular (string).
- Número (entero que contiene un valor único asignado en la instanciación; el primer número de cuenta es 1000).
- Saldo (double).

La clase también incluye los siguientes métodos:

- Credito (permite ingresar dinero en la cuenta).
- Debito (permite retirar dinero de la cuenta).
- ConsultaSaldo (muestra toda la información de la cuenta).

Añada una clase CuentaBancariaRemunerada que representa una cuenta bancaria remunerada que hereda de la clase creada anteriormente y, por tanto, el constructor recibirá como argumentos el nombre del titular y el porcentaje de remuneración de la cuenta.

Redefina el método Credito de CuentaBancariaRemunerada para que la cantidad ingresada aumente en el porcentaje de remuneración definido en el constructor. No es necesario soñar, este funcionamiento bancario atípico es solo para simplificar el ejercicio.

Codifique en el Main una secuencia que permita comprobar el funcionamiento de las clases.

2. Corrección

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LabCuentaBancaria
{
  class Program
  {
    static void Main(string[] args)
    {
      CuentaBancaria cb1 = new CuentaBancaria("Victor");
      cb1.Credito(1000);
      System.Console.WriteLine(cb1.ConsultaSaldo());
      cb1.Debito(200);
      System.Console.WriteLine(cb1.ConsultaSaldo());

      CuentaBancariaRemunerada cb2
                  = new CuentaBancariaRemunerada("González-Aller", 2);
      cb2.Credito(500);
      System.Console.WriteLine(cb2.ConsultaSaldo());

      System.Console.ReadKey();
    }
  }
```

```
class CuentaBancaria
{
  private static int numCuenta = 100;

  public string Titular { get; set; }
  public int Numero { get; set; }
  public double Saldo { get; set; }

  public virtual void Credito(double credito)
  {
    Saldo += credito;
  }

  public void Debito(double debito)
  {
    Saldo -= debito;
  }

  public string ConsultaSaldo()
  {
    return string.Format(
                "Cuenta nº{0} Titular {1} Saldo {2} euros",
                 Numero, Titular, Saldo);
  }

  public CuentaBancaria(string titular)
  {
    this.Titular = titular;
    this.Numero = CuentaBancaria.numCuenta++;
  }
}
class CuentaBancariaRemunerada: CuentaBancaria
{
```

```
    public double PorcentajeRendimiento { get; set; }


    public CuentaBancariaRemunerada(
                  string titular, double porcentajeRendimiento)
     : base(titular)
    {
      PorcentajeRendimiento = PorcentajeRendimiento;
    }


    public override void Credito(double credito)
    {
      base.Credito(credito * (1 + PorcentajeRendimiento / 100));
    }
  }
}
```

Las clases abstractas

Puede suceder que una clase de base contenga métodos imposibles de implementar porque en su nivel en la herencia no disponga de la información necesaria.

Por ejemplo, una clase de base FormaGeometrica ofrece un método virtual Diseñar. A continuación, esta clase se heredada por las clases Triangulo, Rectangulo y Circulo, que van a implementar cada una su propio método Diseñar.

Por lo tanto, la implementación del método Diseñar en la clase de base FormaGeometrica no tiene ningún sentido porque cada forma es específica y, al nivel de FormaGeometrica, esta forma es abstracta.

Entonces, en este caso, ¿por qué definir el método Diseñar en FormaGeometrica?

Esto está relacionado con el polimorfismo. Imagine que está construyendo una aplicación de diseño y que esta aplicación gestiona una serie de formas geométricas cuidadosamente definidas por el usuario. Puede gestionar una lista de objetos de tipo Triangulo, una lista de objetos de tipo Rectangulo, etc. Buena suerte, porque esto será bastante duro. Construyendo una lista de objetos de tipo FormaGeometrica, puede rellenarla de objetos de tipos Triangulo, Rectangulo y Circulo. ¿Por qué? Porque heredan los tres de la clase de base FormaGeometrica y cualquier clase derivada puede implícitamente convertirse en un objeto de su clase de base. Por tanto, todo se simplifica. Cuando haya que diseñar el contenido de la lista, bastará con acceder a cada objeto de tipo FormaGeometrica y llamar a su método Diseñar. Gracias al mecanismo de virtualización estudiado antes, se llamará a la

implementación específica. Por tanto, habrá que definir el método Diseñar en FormaGeometrica para permitir este funcionamiento.

¿Por qué definir una clase abstracta en lugar de una interfaz?

En el ejemplo anterior, la clase FormaGeometrica se habría podido sustituir por una interfaz. Es preferible el uso de una clase abstracta cuando el análisis implique implementar un conjunto de métodos comunes a todas las clases. Recordemos que una interfaz no contiene código y que, si debe mezclar métodos abstractos y reales, la solución que se impone es la clase abstracta.

Por el contrario, si no hay código compartido, es preferible cambiar la clase abstracta que solo contiene los métodos abstractos por una interfaz.

Una clase se debe declarar abstracta tan pronto como contenga un método abstracto.

Es imposible instanciar directamente una clase abstracta. Esto provoca un error de compilación.

Sintaxis de declaración de una clase abstracta:

```
abstract class NombreClaseAbstracta
{
   abstract visibilidad tipoRetorno NombreMetodo ([argumentos]);
   //...
}
```

Los métodos de extensión

Para explicar el interés de los métodos de extensión, nada mejor que un caso concreto. En su aplicación, recibe variables de tipo String y normalmente necesita saber si su contenido son expresiones de valores numéricos.

Este tipo de operaciones no existe en la clase String y como no tiene su código fuente, no la puede añadir.

No puede construir sin más una clase heredada StringEx, porque la clase String es cerrada. Incluso si no lo fuera, hubiera necesitado sustituir todos los tipos String por StringEx en las clases de su proyecto, lo que es fastidioso.

Última solución: escribir el método y ubicarlo en una clase, preferiblemente de tipo static (comúnmente llamada un helper), que utilizará en las ocasiones adecuadas. Este método, muy utilizado, sigue siendo menos práctico que una implementación nativa de su operación en la clase string.

Existe una funcionalidad en C# que permite añadir "sobre la marcha" métodos a los tipos compilados justo en el contexto de su aplicación. No es necesario tener el código fuente del tipo (clase, estructura o interfaz) para añadir el método que le falta; los métodos de extensión sirven para ofrecerle una nueva flexibilidad de codificación. En la parte del código cliente, no hay ninguna diferencia entre la llamada a un método "nativo" y la llamada a un método "de extensión".

Los métodos de extensión se introdujeron en .NET en su version 3.5. Se crearon para LINQ, que es un sistema de consultas genial, fusionado con la gramática C#, lo que permite consultar prácticamente cualquier tipo de colecciones. Las extensiones añadidas sobre la marcha tienen los tipos existentes que se derivan de los operadores para LINQ.

Sintaxis de un método de extensión:

```
public static class nombreClase
{
  public static tipoRetorno nombreMetodo(this tipo instanciaTipo
     [,otros argumentos] )
  {
     //...
  }
}
```

El método se define en una clase de tipo public static y su nombre generalmente se corresponde con el del tipo en el que queremos añadir el método seguido por Helper (por ejemplo, StringHelper).

El método añadido es de tipo static. El inicio de su definición es muy clásico: se define el tipo de retorno y el nombre del método. Después, los argumentos empiezan por la palabra clave this (otra vez él) seguida del tipo en el que se va a añadir el método y, para terminar, una variable que se corresponde con una instancia del mismo tipo que el del método que va a extender.

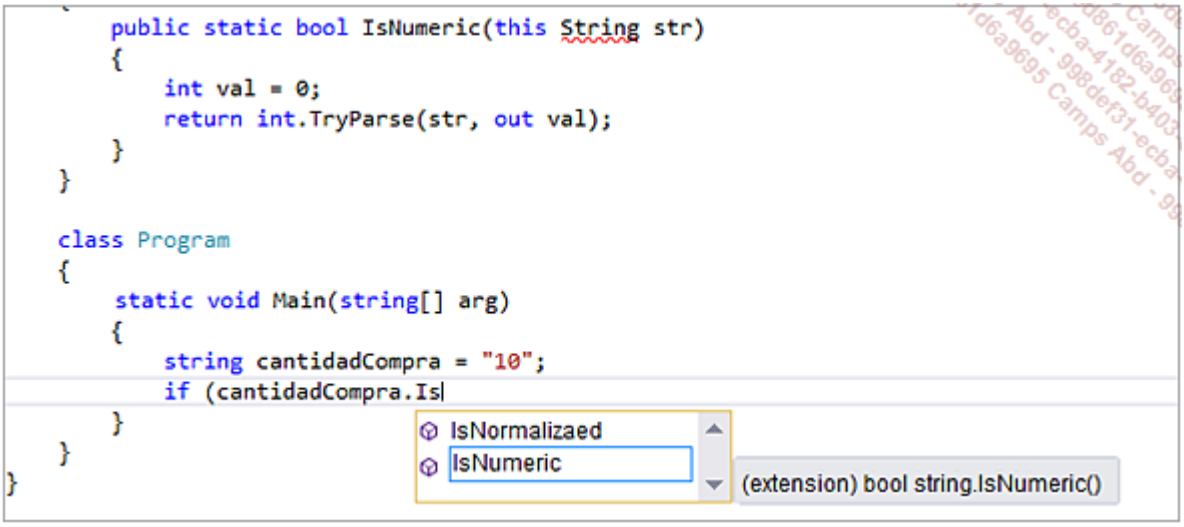
Esta primera variable es muy importante porque el método de extensión va a poder acceder a los miembros "nativos" del objeto a través de él.

Ejemplo:

```
namespace DemoMetodoExtension
{
```

```
  public static class StringHelper
  {
    public static bool IsNumeric(this String str)
    {
      int val = 0;
      return int.TryParse(str, out val);
    }
  }
}
```

En todo el espacio de nombres DemoMetodoExtension los tipos String tendrán sus miembros enriquecidos con un nuevo método: IsNumeric. Además, IntelliSense lo sabe y ofrece todo para indicarle que es una extensión.



Para los demás espacios de nombres, no se pierde nada, basta con utilizar un:

```
using DemoMetodoExtension;
```

como prefijo en el archivo de código fuente para que la extensión se active.

Un método de extensión no puede sustituir a un método nativo, porque extender no es heredar.

La definición de métodos de extensión es una solución elegante para extender las funcionalidades de las clases de las que no tenemos el código fuente. Por el contrario, su uso se desaconseja para extender clases de las que sí tenemos el código fuente. Si desea evolucionar un ensamblado añadiendo nuevos métodos y manteniendo un conjunto de programas compatibles con la versión anterior se aconseja conservar la versión actual en

producción y crear una nueva con el mismo nombre pero con un número de versión diferente. Los nuevos desarrollos que utilizarán esta nueva operativa harán referencia a este nuevo número de versión sin tener que cambiar el nombre del ensamblado referenciado. Gran progreso en comparación a C/C++.

El polimorfismo

1. Entender el polimorfismo

En programación orientada a objetos, el polimorfismo permite a una clase heredada presentarse a una operación como su clase de base o como una de sus interfaces. Gracias a la virtualización de los métodos, la operación que llama al método básico se "enruta" en la clase heredada. De esta manera se obtiene una "especialización" de la operación.

Cualquier referencia a una instancia de clase derivada se puede convertir implícitamente en una referencia a una instancia de tipo de su clase de base.

En C#, la herencia y sus palabras clave virtual y override que se han detallado en el capítulo anterior por una parte permiten a la clase de base definir los métodos que se pueden especializar por parte de sus heredadas y, por otro lado, a su o sus heredadas tener en cuenta los métodos para especializarlos.

En C#, cada tipo es "polimorfo". En efecto, se puede considerar como su propio tipo o como el de su clase de base y, por encadenamientos sucesivos, como el tipo raíz de todos los tipos, a saber System.Object.

2. Explotación del polimorfismo

La mejor programación es la que favorece la independencia entre los módulos. Una arquitectura compuesta de loosely coupled modules es la más recomendada, porque permite la evolución de las capas independientemente del funcionamiento global. Lo que ayer parecía como un bloque monolítico, hoy se convierte en algo fragmentado en niveles (arquitectura n tiers), pudiendo estar repartidos en ubicaciones geográficas totalmente diferentes.

El polimorfismo permite intercambiar módulos. Cada capa acepta módulos que respeten la compatibilidad, además de una lista de comportamientos obligatorios. Las instancias de clases muy tipadas se convierten en objetos "compatibles" con el "contrato" de las interfaces, y la operación principal solo instancia e interconecta las instancias compatibles con estas interfaces.

Cualquier objeto que implemente una interfaz se puede convertir en el tipo de esta interfaz.

En sus clases, dé prioridad a los miembros de tipo "referencia a las interfaces" en lugar de a los miembros de tipo "referencia a objetos". Construirá un código más abierto a las evoluciones de sus componentes.

3. Los operadores is y as

Los operadores is y as permiten comprobar y considerar los objetos básicos como objetos de clases heredadas fuertemente tipadas. Su uso se debe pensar mucho. Si prototipa un método con un argumento débilmente tipado (como System.Object) y debe realizar una operación específica para un tipo de clase heredada concreta, puede utilizar el operador is para comprobar la pertenencia al tipo de la clase heredada y después el operador as para crear una nueva referencia al tipo de la clase heredada y, de esta manera, simplificar su codificación. También puede utilizar directamente el operador as, que devolverá null si el objeto comprobado no es compatible con la clase deseada. El uso directo de as generará un código más rápido en la ejecución, porque no hay una operación de cast. En una consulta sencilla no habrá consecuencias; por el contrario, en una operación repetitiva el impacto será importante.

El evento: estar a la escucha

Igual que los sistemas operativos basados en eventos, puede ser que sus objetos tengan que escuchar a otros objetos. Por ejemplo, su formulario gráfico estará "atento" a las acciones del ratón para poder reaccionar inmediatamente a las peticiones del usuario. Su formulario no sabe cuándo el usuario va a hacer clic en este o aquel componente. De hecho, será el componente el que le llame para indicarle un cambio de estado.

Un objeto que capta la información la puede enviar a una lista de "clientes", que inicialmente se habrán suscrito a su lista de notificación.

La programación orientada a objetos es muy próxima a la realidad. Puede que sea suscriptor a su revista favorita, como miles de otros lectores. Los periodistas escriben artículos que se agrupan en ediciones del periódico. Este periódico se le envía cada cierto tiempo. En cualquier momento puede cancelar su suscripción o suscribirse a otro.

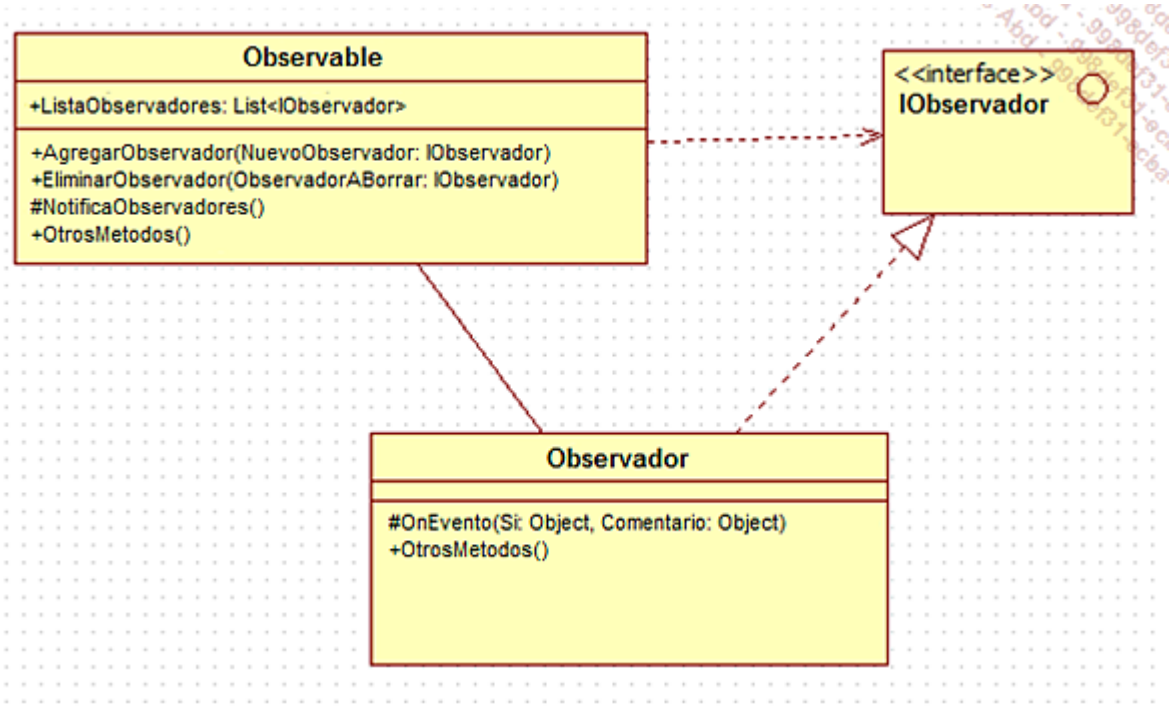
En programación es igual. Habitualmente se trata de establecer mecanismos de gestión de la lista de suscriptores y de las notificaciones.

El pattern Observador

Para este problema recurrente, el "grupo de los cuatro" ofrece un design pattern para resolverlo.

Como vamos a ver, esta solución contrastada se puede utilizar directamente en C#. Sin embargo, existe un sistema con muy buen rendimiento basado en el uso de tipos específicos, que son los delegate y los event, muy utilizados para gestionar las comunicaciones entre objetos en C#.

Pero volvamos al pattern Observador, que ofrece una solución de codificación basada en una relación entre uno (observado) y varios (observadores), utilizando el acoplamiento más débil posible. A continuación se muestra la representación UML de este tipo de relación.



La clase Observable y la interfaz IObservador se diseñan al mismo tiempo, cuando se desarrolla un objeto que puede comunicar sus cambios de estado.

La comunicación entre un objeto Observable, que se ha podido desarrollar más tarde que el resto, y un objeto Observador es posible gracias al polimorfismo del objeto Observador que implementa la interfaz IObservador.

De hecho, la clase Observable solo reconoce objetos que implementan la interfaz IObservador (vea los métodos de gestión de los suscriptores: esperan un objeto de tipo IObservador) y por tanto, cuando una clase implementa esta interfaz, se convierte en "compatible" con las notificaciones de la clase Observable.

A continuación se muestra la implementación en C# del lado de la notificación:

```
    // Interfaz que deberá implementar cualquier clase
    // que quiera observar la clase... Observable :)

interfaz IObservador
    {
```

```
        void ObservableNotificaSusObservadores(object sender, object arg);
    }
    // La clase Observable
    // durante su trabajo, puede notificar cambios
    // a las clases que la observan.
    class Observable
    {
        // Identificador de instancia para la traza
        static int count = 0;
        public int Id { get; set; }
        public Observable() { Id = count++; }
        // A continuación se muestra la lista de observadores
        private List<IObservador> ListaObservadores = new
List<IObservador>();
        // El método que permite añadir un observador a la lista
        // Note cómo se admite el observador si implementa IObservador
        public void NewObservador(IObservador s)
        {
            ListaObservadores.Add(s);
        }
        // Un cambio interviene, por lo que la clase notifica
        // a todos sus observadores
        public void CambioObservable(object arg)
        {
            foreach (IObservador item in ListaObservadores)
            {
                item.ObservableNotificaSusObservadores(this, arg);
            }
        }
    }
```

Suscripción/Eliminación de la suscripción

La lista de suscriptores es una lista de instancias de objetos que implementan la interfaz IObservador. Aquí se utiliza una lista genérica, que se ha estudiado más atrás. Dos métodos permiten añadir y eliminar entradas en esta lista.

Notificación

La clase Observable llama internamente al método CambioObservable. Imaginemos que se detecta un cambio de estado en el elemento que encapsula. Invoca entonces a este método para informar a todos sus suscriptores. Como argumento de este método envía un objeto cuya función es la de "comentar" el evento. En este fragmento, el argumento es de tipo Object pero podría ser fuertemente tipado y definirse en el mismo namespace que la clase Observable. El método CambioObservable realiza una iteración en la lista y, para cada instancia, llama al método obligatoriamente implementado de IObservador, a saber ObservableNotificaSusObservadores.

Observe que ObservableNotificaSusObservadores recibe dos argumentos. No es obligatorio, pero generalmente es lo más práctico. El primero representa la instancia del Observable. De hecho, un observador puede observar a varios objetos de tipo Observable y, por lo tanto, debe poder diferenciarlos, porque el método ObservableNotificaSusObservadores será el mismo para todos. El segundo argumento es el objeto que describe la notificación y que se construyó por la clase Observable durante la detección del cambio de estado. Este objeto debe contener toda la información del evento. Por ejemplo, si es un receptor el que envía un cambio de estado de una de sus entradas, deberá buscar el número de la entrada en cuestión y su nuevo estado.

A continuación se muestra la implementación del código del lado del Observador:

```
// La clase Observador
    // Se puede haber diseñado después de Observable
    // Para hacerla "compatible" con Observable, basta
    // con que implemente la interfaz IObservador
    class Observador: IObservador
    {
        // Identificador de instancia para la traza
        static int count = 0;
        int id;
        public Observador() { id = count++; }


        public void NewObservable(Observable o)
        {
            o.NewObservador(this);
        }
```

```
        // El método al que Observable va a llamar
        public void ObservableNotificaSusObservadores(object
sender, object arg)
        {
            // y que va a mostrar una línea en la consola
            Console.WriteLine("Observador {0} notificado por
Observable {1}",
                              id, (sender as Observable).Id);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Creación de un observable.
            Observable observable = new Observable();
            // y de diez observadores
            Observador[] tabObservadores = new Observador[10];
            for (int i = 0; i < tabObservadores.Length; i++)
            {
                Observador t = new Observador();
                t.NewObservable(observable);
                tabObservadores[i] = t;
            }
            // ...
            // Simulación de un cambio en el observable
            observable.CambioObservable(new object());

            Console.ReadKey();
        }
    }
```

La clase Observador implementa la interfaz IObservador, haciéndola compatible con la escucha de los objetos de tipo Observable. El método NewObservable permite establecer la unión entre Observador y Observable. Por último, se llama al método ObservableNotificaSusObservadores cuando Observable tiene algo que decir.

La clase Program instancia un objeto Observable y le conecta diez objetos Observadores. Para comprobar el funcionamiento, "simula" un cambio de estado del Observable. Observable y Observador tienen un número asignado en su construcción que permite mostrar los mensajes significativos.

A continuación se muestra la salida por la consola correspondiente:

```
Observador 0 notificado por Observable 0

Observador 1 notificado por Observable 0

Observador 2 notificado por Observable 0

Observador 3 notificado por Observable 0

Observador 4 notificado por Observable 0

Observador 5 notificado por Observable 0

Observador 6 notificado por Observable 0

Observador 7 notificado por Observable 0

Observador 8 notificado por Observable 0

Observador 9 notificado por Observable 0

 Pulse una tecla para continuar...
```

La solución C#: delegate y event

En lenguaje C, la comunicación entre módulos, por ejemplo entre un ejecutable y una DLL, se hace por medio de un puntero de función.

El puntero de función es una ubicación en memoria que contiene la dirección de la función a ejecutar. El sistema operativo permite al programa cambiar una DLL y después recuperar dinámicamente la dirección de la función que exporta a partir de su nombre. Después, gracias a una sintaxis adecuada, el programa llama a la función como si fuera "local".

El problema de esta solución es que un puntero de función no informa ni sobre el nombre, ni sobre el tipo de los argumentos, ni sobre el valor de retorno. Un puntero de función está compuesto por solo cuatro bytes, que es el origen de muchos funcionamientos incorrectos.

En el mundo de C#, el desarrollador dispone de un tipo especial llamado delegate que va a suprimir este defecto tan importante de C yendo mucho más lejos.

Un delegate es un objeto:

- que define un prototipo de método de tipo static o instancia, con sus argumentos y su tipo de retorno.
- que contiene una colección de referencias a los métodos compatibles con su prototipo.

- que se puede pasar como argumento a otro método.

Sintaxis de un delegate:

```
visibilidad delegate tipoRetorno NombreDelegado([tipo
argumento nombre argumento,...]);
```

Ejemplo:

```
using System;

namespace DemoDelegado
{
  // Este delegate puede apuntar a cualquier
  // método que reciba dos enteros como argumento
  // y devuelva un entero
  public delegate int Calcular(int x, int y );

  // Esta clase contiene los métodos
  // "compatibles" con el delegate Calcular
  public class CalculoBasico
  {
    public static int Agregar(int x, int y )
    { return x + y ; }
    public static int Sustraer(int x, int y )
    { return x - y ; }
  }

  class Program
  {
    static void Main(string[] args)
    {
      // Creación de un delegate Calcular que apunta
      // a CalculoBasico.Agregar().
      Calcular b = new Calcular(CalculoBasico.Agregar);
```

```
      // Invoca al método CalculoBasico.Agregar()
      // pasando por el delegate
      int resultado = b(10, 10);
      Console.WriteLine("10 + 10 = {0}", resultado);



      // Invoca al método CalculoBasico.Sustraer()
      // pasando por el delegate
      b = CalculoBasico.Sustraer;
      resultado = b(10, 10);
      Console.WriteLine("10 - 10 = {0}", resultado);
    }
  }
}
```

En este fragmento de código, un tipo delegate llamado Calcular se define directamente en el espacio de nombres. Observe que se hubiera podido definir en una clase. A continuación, se declara una clase CalculoBasico que implementa dos métodos a las firmas compatibles con este delegate.

Una clase Program instancia un objeto de tipo delegate Calcular, asociándole el método Agregar de la clase CalculoBasico.

El hecho de llamar a la instancia del delegate con los argumentos como si se tratara de un método clásico provoca la llamada al método Agregar.

En realidad, es un acceso directo de escritura lo que nos ofrece C#. Esta sintaxis llama implícitamente al método Invoke del objeto delegate. Invoke provoca una llamada síncrona al método al que apunta. Una llamada síncrona significa que la ejecución es bloqueante.

```
b(10,10);
// es equivalente a
b.Invoke(10, 10);
```

A continuación, el objeto delegate se reconfigura para contener una referencia al método Sustraer de la clase CalculoBasico y el método se llama de la misma manera.

A continuación se muestran la salida de este encadenamiento por la consola.

```
10 + 10 = 20
10 - 10 = 0
```

```
Pulse una tecla para continuar...
```

1. Utilización del delegate en el design pattern Observador

Es posible utilizar el principio del delegate para que se comuniquen los observables con los observadores. El delegate, definido por el diseñador de la clase observable, se ajusta a la firma del método que cada observador deberá implementar si quiere recibir las notificaciones observables.

Además, el tipo delegate integra una funcionalidad multicast que es capaz de gestionar una lista de métodos compatibles con su firma y, finalmente, una lista de observadores suscriptores.

El registro de un suscriptor se hace de una manera muy práctica mediante la sobrecarga del operador += y la eliminación de la suscripción por su homólogo, el operador -=.

A continuación se muestra un fragmento de código de la clase Observable que utiliza un delegate:

```
using System;


namespace DemoObservadorObservableDelegado
{
    // Definición de un delegate que prototipa un método
    // que se llama cuando se produce un cambio
    // Los tipos de los argumentos se podrán modificar
    delegate void delegateCambio(object sender, object arg);


    // La clase Observable
    // Durante su trabajo, puede notificar sus cambios
    // a las clases que la observan.
    class Observable
    {
        // Identificador de instancia para la traza
        static int count = 0;
        public int Id {get; set;}
        public Observable() { Id = count++; }
```

```
        // Variable que va a recibir el objeto delegate
        private delegateCambio DelegateCambia;

        // Método de registro de observadores
        public void NewObservador(delegateCambio d)
        {
            // Si es la primera llamada durante la instanciación
            // del delegate
            if (DelegateCambia == null)
                DelegateCambia = new delegateCambio(d);
            else
                DelegateCambia += d;
        }

        // Se produce un cambio, de modo que la clase notifica
        // a todos sus observadores
        public void CambioObservable(object arg)
        {
            if (DelegateCambia != null)
                DelegateCambia(this, arg);
        }
    }
```

A continuación se muestra el fragmento de código de lado del observador y del lado del programa:

A diferencia de lo que sucede con el enfoque por interfaz (enfoque "muy Java"), el observador no necesita heredar de nada. Basta con que implemente un método que se corresponda con la definición del delegate (OnNotification en el siguiente código) del observable y pasar su referencia al método de suscripción.

```
    // La clase Observador
    // Se puede haber diseñado después de Observable
    class Observador
    {
        // Identificador de instancia para la traza
        static int count = 0;
        int id;
        public Observador() { id = count++; }

        public void NewObservable(Observable o)
        {
            o.NewObservador(ObservableNotificaSusObservadores);
        }

        // El método que Observable va a llamar
        public void ObservableNotificaSusObservadores(object sender, object arg
)
        {
            // y que va a mostrar una línea en la consola
            Console.WriteLine("Observador {0} notificado por
Observable{1}",
                id, (sender as Observable).Id);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Creación de un observable.
            Observable observable = new Observable();
            // y de diez observadores
            Observador[] tabObservadores = new Observador[10];
            for (int i = 0; i < tabObservadores.Length; i++)
```

```
            {
                Observador t = new Observador();

                t.NewObservable(observable);
                tabObservadores[i] = t;
            }
            // ...
            // Simulación de un cambio en el observable
            observable.CambioObservable(new object());

            Console.ReadKey();
        }
    }
```

La clase Program instancia un objeto observable y le conecta diez observadores. Para comprobar el funcionamiento de la notificación, simula un evento llamando al método NotificaObservadores del objeto Observable.

A continuación se muestra la salida por la consola asociada:

```
Observador 0 notificado por Observable 0
Observador 1 notificado por Observable 0
Observador 2 notificado por Observable 0
Observador 3 notificado por Observable 0
Observador 4 notificado por Observable 0
Observador 5 notificado por Observable 0
Observador 6 notificado por Observable 0
Observador 7 notificado por Observable 0
Observador 8 notificado por Observable 0
Observador 9 notificado por Observable 0
```

2. Utilización de un evento

El delegate es un medio potente para establecer la comunicación entre objetos. Sin embargo, hay que verificar que se respetan las reglas de encapsulación y hacerlo no accesible "desde el

exterior". De hecho, su uso indebido por medio de una operación "maliciosa" podría tener consecuencias terribles sobre el plan de la seguridad. La lista de los suscriptores se podría utilizar para enviar información errónea o esta misma lista se podría sustituir por otra lista de entidades no autorizadas.

En el ejemplo anterior, el código de registro y borrado de la suscripción se realiza por métodos públicos de la clase Observable, y el delegate es de tipo private. Por tanto, no hay un problema de seguridad, aunque el código del observable es un poco "pesado".

C# ofrece una alternativa con la palabra clave event.

Un event es una palabra clave que se asocia a un tipo de delegate particular y a un nombre de miembro de instancia. Cuando el compilador encuentra esta palabra clave, define un juego de métodos que simplifican y securizan el uso del delegate.

Sintaxis de un evento

```
[visibilidad] evento tipo nombre;
```

La mayor parte de las veces, el atributo de visibilidad de un event es de tipo public, porque los observadores lo van a utilizar directamente.

El tipo es el nombre del delegate a gestionar. El event se debe declarar en una clase, porque encapsula la instancia del delegate (mientras que el delegate es una definición y se puede escribir fuera de una clase).

Del lado de la seguridad, observe que el event de tipo public limita las operaciones sobre la lista de suscriptores del delegate a la adición de nuevos clientes y a su eliminación. Sin el event, un delegate 'public' ofrecería métodos mucho más permisivos, como la inicialización completa de la lista de los suscriptores, etc.

A continuación se muestra el código anterior adaptado al uso de un event. Del lado del Observable, la clase no tiene más métodos de gestión de sus observadores. Del lado del Observador, el código de registro se simplifica por el uso del operador +=.

```
using System;


namespace DemoObservadorObservableDelegadoEvento
{
    // Definición de un delegate que prototipa un método
    // que se llama cuando se produce un cambio
    // Los tipos de los argumentos se podrán modificar
    delegate void delegateCambio(object sender, object arg);


    // La clase Observable
    // Durante su trabajo, puede notificar sus cambios
    // a las clases que le observan.
    class Observable
```

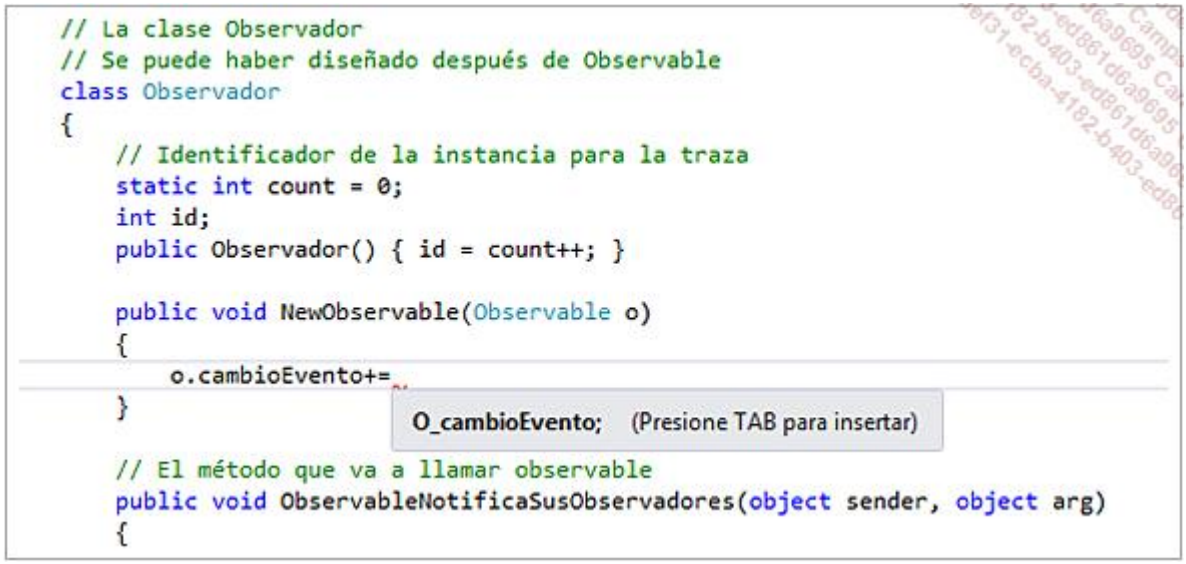
```
{
    // Identificador de instancia para la traza
    static int count = 0;
    public int Id { get; set; }
    public Observable() { Id = count++; }

    // Evento asociado al cambio
    public event delegateCambio cambioEvento;

    // Se produce un cambio, por lo que la clase notifica
    // a todos sus observadores
    public void CambioObservable(object arg)
    {
        if (cambioEvento != null)
            cambioEvento(this, arg);
    }
}

// La clase Observador
// Se puede haber diseñado después de Observable
class Observador
{
    // Identificador de la instancia para la traza
    static int count = 0;
    int id;
    public Observador() { id = count++; }

    public void NewObservable(Observable o)
    {
        o.cambioEvento += ObservableNotificaSusObservadores;
    }
    // El método que va a llamar observable
    public void ObservableNotificaSusObservadores
```

```
(object sender, object arg)
        {
            // y que va a mostrar una línea en la consola
            Console.WriteLine("Observador {0} notificado
por Observable {1}",
                id, (sender as Observable).Id);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Creación de un observable.
            Observable observable = new Observable();
            // y de diez observadores
            Observador[] tabObservadores = new Observador[10];
            for (int i = 0; i < tabObservadores.Length; i++)
            {
                Observador t = new Observador();
                t.NewObservable(observable);
                tabObservadores[i] = t;
            }
            // ...
            // Simulación de un cambio en observable
            observable.CambioObservable(new object());
            Console.ReadKey();
        }
    }
}
```

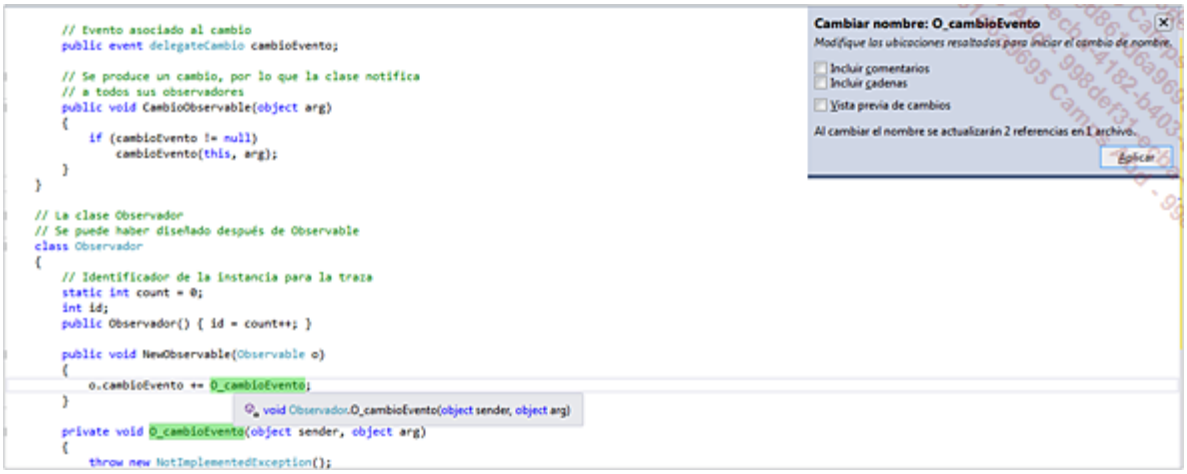
El entorno de desarrollo Visual Studio da asistencia al desarrollador con el uso de event de las clases Observable.

Imagine, mientras escribe el código anterior, que se está codificando el registro de la suscripción.

Escriba el nombre de la referencia del observable y después ".", después el nombre del event al que se desea suscribir y después "+=". Visual Studio le ofrece un nombre de método que va a recibir la notificación.



Pulse la tecla [Tab] y Visual Studio inserta en su archivo de código fuente un método conforme al event que desea utilizar y muestra un cuadro de diálogo para que pueda renombrarlo si el nombre por defecto no le conviene.



Después de haber eliminado la línea que muestra la excepción NotImplementedException puede implementar el cuerpo del método. Es mágico.

3. Cómo acompañar al evento de datos

El prototipo del método gestionado por un delegate puede recibir y devolver cualquier tipo de datos. Los delegate de .NET generalmente ofrecen un primer argumento de tipo System.Object que representa al objeto original de la notificación, y después un segundo argumento heredado o no de System.EventArgs.

Durante la creación de una clase que ofrece delegate se recomienda crear un tipo heredado de System.EventArgs que contenga toda la información útil para "comentar" la notificación.

Ejemplo de un tipo utilizado para acompañar un cambio de nivel de entrada en una tarjeta de entrada/salida:

```
// Clase que acompaña la notificación de cambio de valor
// de una entrada analógica
 public class ChangeAnalogValueEventArgs: EventArgs
 {
     // número de la entrada analógica que se ha modificado
     private int number;
     public int Number
     {
         get { return number; }
         set { if( value <3 && value > 0) number = value; }
     }
     public int value;   // nuevo valor leído (0-255)
 }
```

El objeto heredado de EventArgs puede contener estados y métodos.

La clase se puede definir en la raíz del namespace o estar directamente anidada en la clase Observable por motivos de homogeneidad.

Extracto de código que envía la notificación, con su descriptor específico:

```
void NotifyAnalogInputValueChange(int inputNumber,
                                  int inputValue)
{
```

```
      if (OnAnalogInputValueChange != null)
      {
          OnAnalogInputValueChange(this,
            new ChangeAnalogValueEventArgs
            { Number = inputNumber, value = inputValue }
            );
      }
  }
```

4. Recursos genéricos para simplificar todavía más

Dando por hecho que la mayor parte de los event que hay en .NET encapsulan los delegate, cuya firma tiene:

- un valor de retorno de tipo void.
- un primer argumento de tipo Object, identificando al emisor.
- un segundo argumento opcional que deriva de la clase EventArgs, que da los argumentos del evento.

.NET dispone de una sintaxis que define el segundo argumento como tipo Generic.

Sintaxis:

```
EventHandler[<TEventArgs>] NombreDelegado;
```

TEventArgs se define si el event gestiona los datos durante la notificación. Si no lo hace, entonces se utiliza directamente EventHandler y el tipo del segundo argumento del método del lado del observador será EventArgs. Empty.

Ejemplo de event sin datos:

```
public EventHandler MisObservadores;
```

Ejemplo de método compatible:

```
public void OnNotification(object sender, EventArgs arg){}
```

Si se define TEventArgs, se aconseja que el método del lado del observador esté prototipado para soportar directamente este tipo de objetos.

Ejemplo de event con datos:

```
public EventHandler<ObservableEventArgs> MisObservadores;
```

Ejemplo de método compatible:

```
public void OnNotification(object sender,
                           ObservableEventArgs arg){}
```

5. Las expresiones lambda

Antes: los métodos anónimos

Partiendo de

- que el método del observador solo se utilizará por el observable,
- que el delegate es suficientemente preciso en la firma del método,

los diseñadores de C# buscaron la manera de reducir todavía más el número de líneas de código, introduciendo un concepto de métodos sin nombre cuyo cuerpo se define directamente durante la suscripción al event.

La primera version de esta sintaxis tan particular apareció con la segunda version del lenguaje C#, con la denominación de métodos anónimos.

Sintaxis de un método anónimo:

```
delegate([TipoArgumento1 nombreArgumento1, TipoArgumento2 nombreArgumento2...)
{
   // operaciones
};
```

A continuación se muestra nuestra clase Observador, que desarrolla un método anónimo durante la suscripción al event de Observable.

```
  class Observador
  {
    // Identificador de instancia para la traza
    static int count = 0;
    int id;
    public Observador() { id = count++; }

    public void NewObservable(Observable o)
    {
      // El método llamado por el observable se
      // declara "en línea" con el registro
      o.MisObservadores
```

```
      += delegate(object sender, ObservableEventArgs arg)
    {        // Cuerpo del método anónimo
      Console.WriteLine(
        "Observador {0} notificado por Observable {1}",
            id, (sender as Observable).Id);
    };  // << Importante no olvidar el;
  }
}
```

Consecuencia: el método anónimo solo se puede llamar por event.

Particularidad: el método anónimo, que se llamará después del registro, podrá utilizar los datos locales del método que lo haya definido.

A continuación se muestra una nueva version de la clase Observador, que lo prueba.

```
class Observador
{
  // Identificador de instancia para la traza
  static int count = 0;
  int id;
  public Observador() { id = count++; }
  // Generador de datos aleatorios
  private static Random random;
  private static int GenerateRandomNumber(int max)
  {
    if (random == null)
      random = new Random();
    return random.Next(max);
  }

  public void NewObservable(Observable o)
  {
    // Justo antes de la suscripción, se genera un número
    // aleatorio almacenado en una variable local
```

```
      // del método.
      int intLocalNewObservable
        = Observador.GenerateRandomNumber(20);
      // Mostrar este valor
      Console.WriteLine(
        "NewObservable para Observador {0} random {1}",
        id, intLocalNewObservable);

      // Suscripción/Codificación
      o.MisObservadores
        += delegate(object sender, ObservableEventArgs arg)
        { // Este código se llama "más tarde", durante
          // una notificación del observable
          // pero todavía pudiendo ver
          // el dato intLocalNewObservable
          Console.WriteLine(
            "Observador {0} random {1} notificado",
            id, intLocalNewObservable);
        };
    }
  }
```

La clase Program se modifica ligeramente para insertar una pausa entre las suscripciones de los Observadores y la notificación del Observable.

```
  class Program
  {
    static void Main(string[] args)
    {
```

```
      // Creación de un observable
      Observable observable = new Observable();
      // y diez observadores
      Observador[] tabObservadores = new Observador[10];
      for (int i = 0; i < tabObservadores.Length; i++)
      {
        Observador t = new Observador();

        t.NewObservable(observable);
        tabObservadores[i] = t;
      }
      // Espera una pulsación de una tecla para separar
      // la parte suscripción de la parte notificación
      Console.WriteLine("Pulse una tecla.");
      Console.ReadKey();

      // ...
      // Simulación de un cambio en el observable
      observable.NotificaObservadores(new ObservableEventArgs()
      { ObservableInt = 1 });

      Console.ReadKey();
    }
  }
```

Ahora, la clase Observador contiene un generador static de números aleatorios que permite obtener valores diferentes para cada instancia. Este número aleatorio se guarda en un dato local al método de suscripción. Teóricamente, el ciclo de vida de este número (objeto de tipo Valor) es el del método NewObservable.

Por tanto, va a continuar siendo accesible para el método anónimo, como se muestra en esta traza:

```
NewObservable para Observador 0 random 4
Fin NewObservable
NewObservable para Observador 1 random 5
Fin NewObservable
```

```
NewObservable para Observador 2 random 17
Fin NewObservable
NewObservable para Observador 3 random 11
Fin NewObservable
NewObservable para Observador 4 random 19
Fin NewObservable
NewObservable para Observador 5 random 7
Fin NewObservable
NewObservable para Observador 6 random 3
Fin NewObservable
NewObservable para Observador 7 random 14
Fin NewObservable
NewObservable para Observador 8 random 15
Fin NewObservable
NewObservable para Observador 9 random 13
Fin NewObservable
Pulse una tecla.
Observador 0 random 4 notificado
Observador 1 random 5 notificado
Observador 2 random 17 notificado
Observador 3 random 11 notificado
Observador 4 random 19 notificado
Observador 5 random 7 notificado
Observador 6 random 3 notificado
Observador 7 random 14 notificado
Observador 8 random 15 notificado
Observador 9 random 13 notificado
```

Pequeña restricción: el método anónimo no puede acceder a los argumentos de tipo ref u out del método que lo alberga.

Las expresiones lambda

La tercera versión del lenguaje C# introdujo una nueva sintaxis que simplifica todavía más los métodos anónimos con la llegada del operador =>.

La novedad es no tener que definir más el tipo de los argumentos del método, porque se definen por el delegate y, por lo tanto, "se deducen" por el compilador. Hay que llamarlos para que el cuerpo de la función pueda referenciarlos.

Sintaxis de una expresión lambda:

```
([nombreArgumento1, nombreArgumento2...)=>
{
   // operaciones
};
```

A continuación se muestra la version "expresión lambda" de nuestra clase Observador:

```
  class Observador
  {  // Identificador de instancia para la traza
    static int count = 0;
    int id;
    public Observador() { id = count++; }
    public void NewObservable(Observable o)
    {   // El método llamado por el observable se
      // declara "en línea", con la suscripción
      o.MisObservadores
        += (sender, arg)=>
        {
          Console.WriteLine(
            "Observador {0} notificado por Observable {1}",
            id, (sender as Observable).Id);
        };  // Importante, no olvidar el ;
    }
  }
```

Las expresiones lambda no se limitan al modelo Observador/Observable. Son omnipresentes en .NET y sustituyen ventajosamente a los métodos anónimos.

Menos código, pero más difícil de leer

Los métodos anónimos y las expresiones lambda no gozan de opiniones unánimes entre los desarrolladores, sobre todo cuando deben pelearse con varios lenguajes. Hay que reconocer que este tipo de sintaxis ocupa menos líneas de código, ofreciendo más flexibilidad de memoria, pero su lectura se puede hacer bastante difícil.

Desde C# 6, es posible utilizar una expresión lambda dentro del get de una propiedad.

Como por ejemplo, en el siguiente código para calcular una cantidad con iva:

```
class DePrueba
{
    public double SinImpuestos { get; set; }
    public double IVA { get; set; }

    public double ConIva => SinImpuestos * (1 + IVA / 100);
}

class Program
{
    static void Main(string[] args)
    {
        var miClaseDePrueba = new DePrueba();
        miClaseDePrueba.SinImpuestos = 10.35;
        miClaseDePrueba.IVA = 21;
        Console.WriteLine(miClaseDePrueba.ConIva);
    }
}
```

6. Ejemplo de uso de event

Ahora que tenemos una visión de conjunto de los delegate y los event, es momento de ver cómo .NET los explota para gestionar los eventos en los formularios gráficos. Para esto, vamos a utilizar la interfaz gráfica .NET Windows Forms para construir nuestros formularios de prueba.

Windows Forms no es la única solución para crear interfaces gráficas en .NET. También tenemos Windows Presentation Foundation (WPF) y Silverlight.

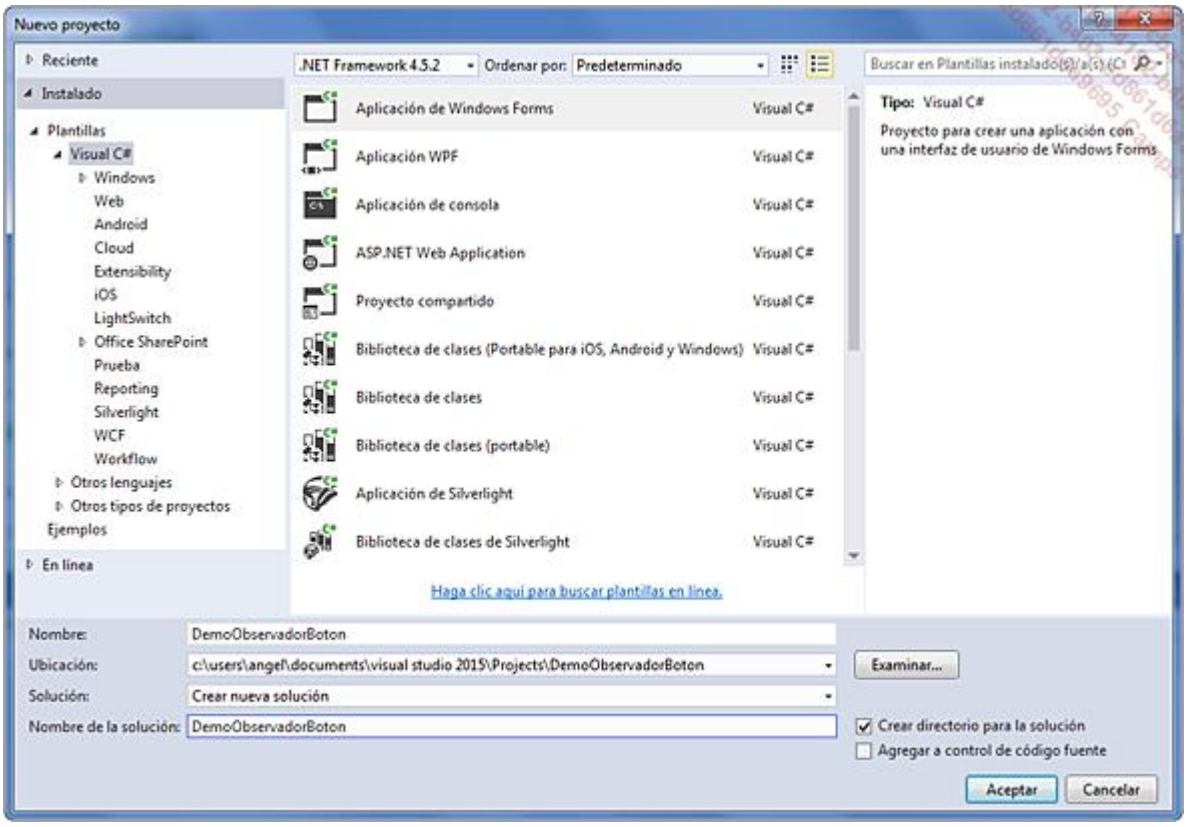
WPF apareció con la version 3.0 de .NET. Hasta ese momento, el desarrollador debía hacer malabares con las interfaces de programación (API) de Windows Forms y GDI+ para construir interfaces gráficas "sofisticadas". Windows Presentation Foundation aporta una solución de programación unificada. Las interfaces gráficas se construyen con un lenguaje de gramática XML (archivo XAML) con el que los diseñadores gráficos pueden trabajar con una herramienta adaptada, como Microsoft Blend. El desarrollador dispone en Microsoft Visual Studio de un editor gráfico XAML, menos potente pero suficiente para hacer el enlace con las clases C#. Durante la ejecución, WPF aprovecha de manera satisfactoria la aceleración del hardware de la máquina.

Silverlight es una implementación reducida de Windows Presentation Foundation, de una CLR y de las clases .NET. El runtime Silverlight funciona en los navegadores de Internet y permite así el funcionamiento de aplicaciones .NET en diferentes plataformas (como Mac OS X).

Por tanto, vamos a dejar la aplicación de tipo consola, para fabricar un formulario basado en la tecnología Windows Forms. Este formulario va a mostrar un botón y cuando se pulse en él, se mostrará una ventana "Hello World!". Esta no será la aplicación del siglo, pero nos va a permitir entender el uso de event para gestionar eventos.

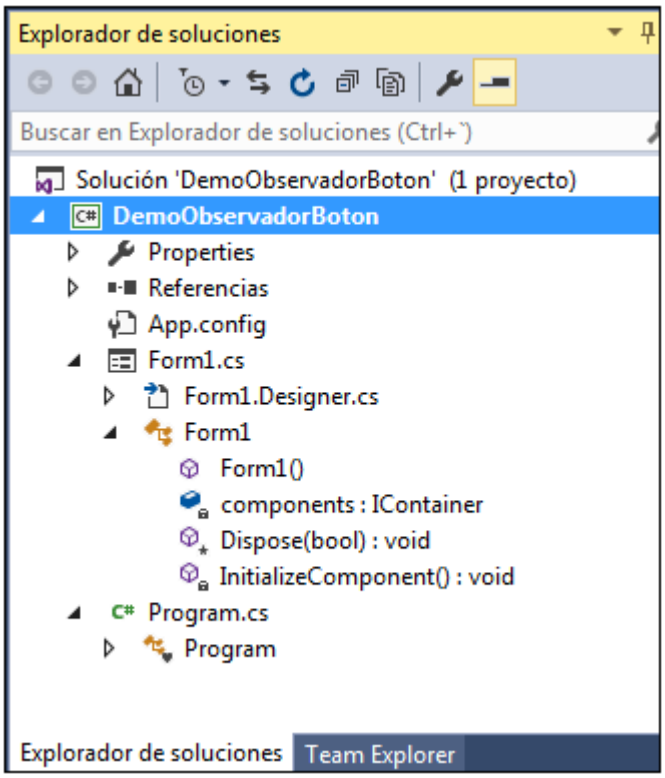
En el menú **Archivo**, seleccione **Nuevo Proyecto**.

Seleccione en el formulario el modelo **Visual C#** para **Aplicación de Windows Forms**. Indique **DemoObservadorBoton** como nombre de la solución.



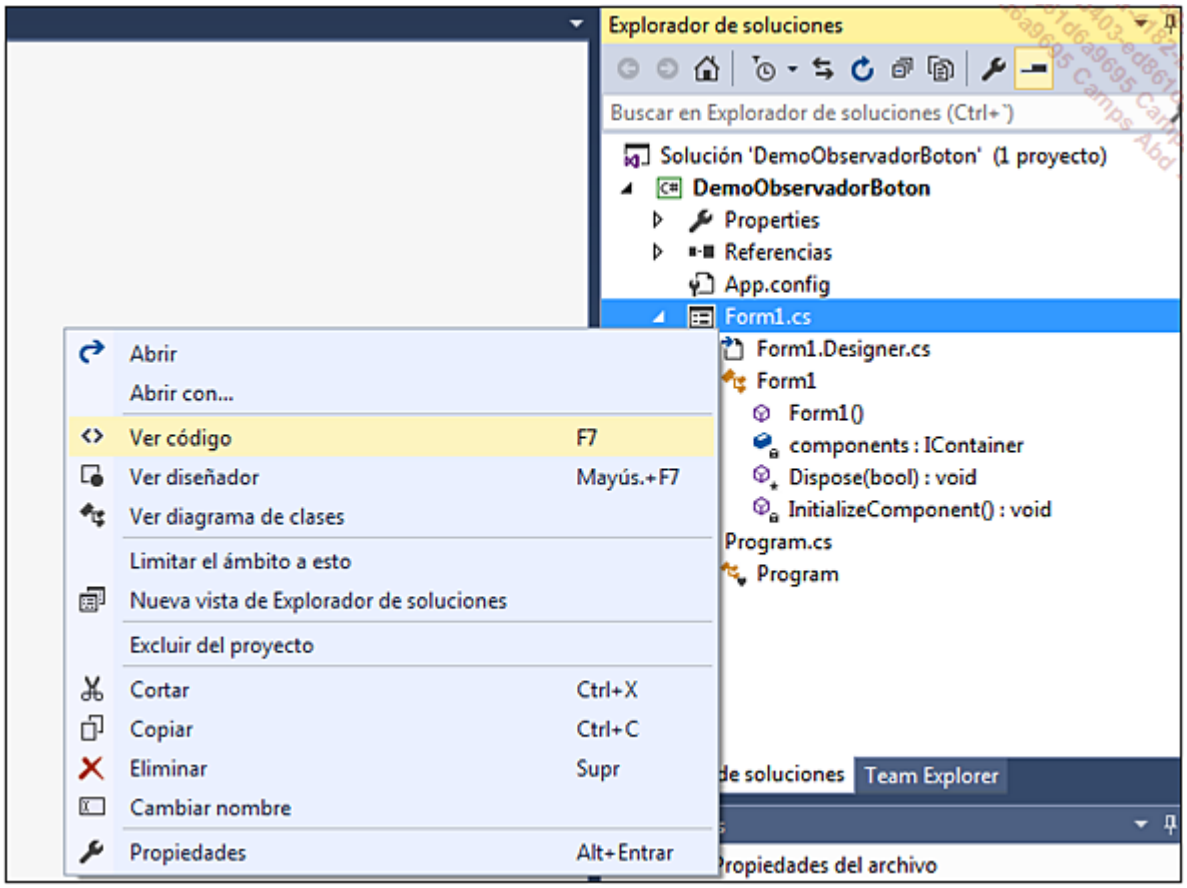
Visual Studio abre el editor gráfico Windows Forms. Este editor permite disponer de los controles gráficos dentro del formulario y, sobre todo, crear el código C# que permite utilizarlos. El archivo que contiene la clase por defecto que encapsula el formulario se

llama **Form1.cs**. Puede acceder a ella desde el explorador de soluciones, situado a la derecha.



Para pasar del editor gráfico al editor de código, basta con pulsar la tecla de función [F7] o hacer clic en el botón derecho del ratón sobre **Form1.cs** y seleccionar **Ver código**.

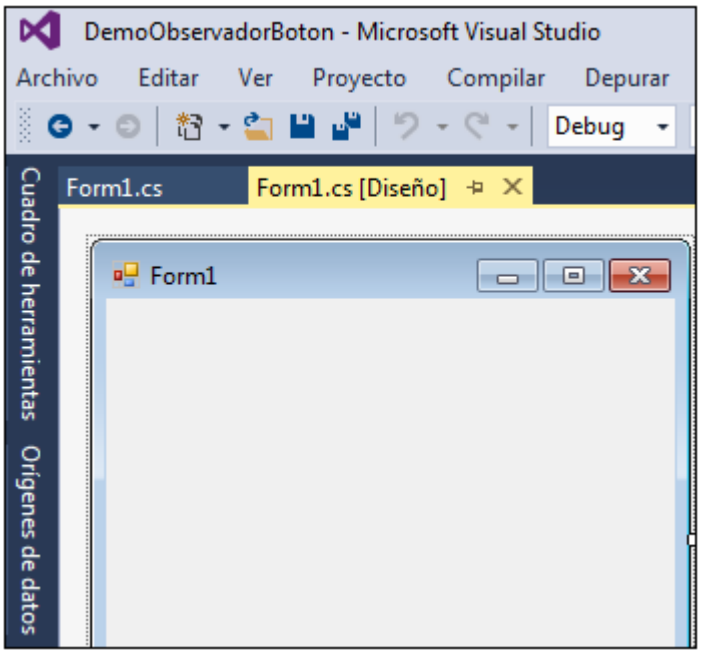
Para pasar del editor de código al editor gráfico, basta con pulsar la combinación de teclas [Mayus][F7] o hacer clic en el botón derecho del ratón en **Form1.cs** y seleccionar **Diseñador de vistas**.



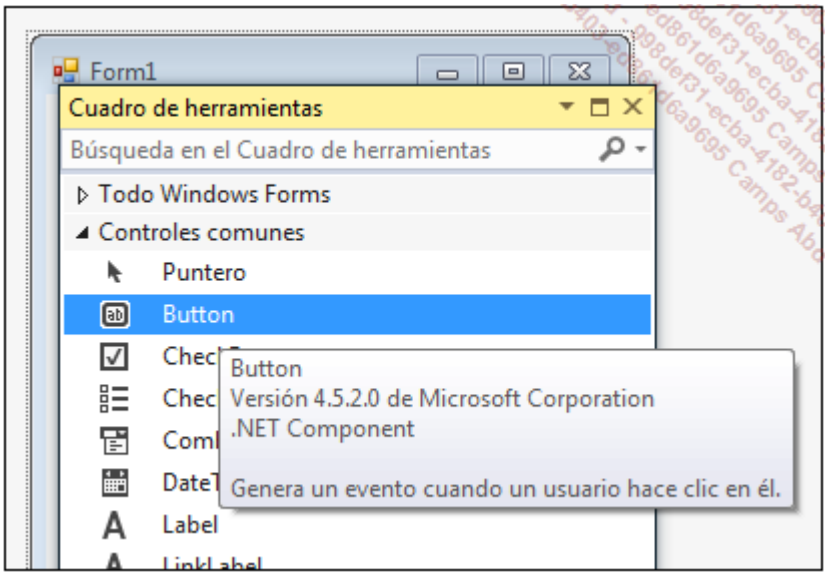
La clase Form1 se define en dos archivos fuente, para que el editor gráfico pueda disponer de un contenido "parcial" modificable, independientemente del código del desarrollador

(revise la presentación anterior sobre la palabra clave partial). Es el compilador el que recoge los archivos fuente para formar la clase completa.

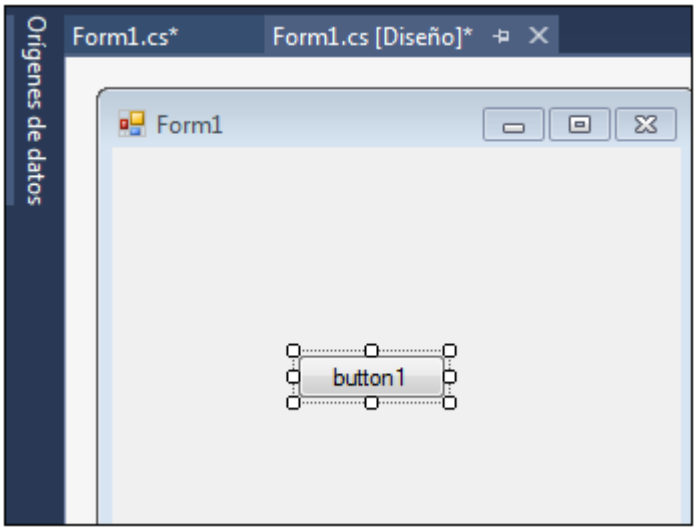
Llame al **Cuadro de herramientas** en la parte superior izquierda de la pantalla.



Seleccione el control **Botón** dentro de **Controles comunes**.



Después arrástrelo al formulario Form1.



El asistente ha modificado la clase Form1 (con el archivo Form1.Designer.cs) en consecuencia, añadiendo una referencia privada a un objeto de tipo Button.

```
private System.Windows.Forms.Button button1;
```

A continuación ha añadido al método InitializeComponent, la instanciación del botón y el ajuste de su ubicación en el formulario.

```
  private void InitializeComponent()
  {
    this.button1 = new System.Windows.Forms.Button();
    this.SuspendLayout();
    //
    // button1
    //
    this.button1.Location= new System.Drawing.Point(70, 43);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 0;
    this.button1.Text = "button1";
    this.button1.UseVisualStyleBackColor = true;
    //...
  }
```

Ahora vamos a utilizar un event para que el clic en este botón llame a un método de la clase Form1. El editor gráfico de Visual Studio nos va a ayudar completamente en esta

tarea, porque es suficiente con hacer doble clic en el control button1 en el formulario para que se prepare el soporte al evento "clic izquierdo" en este botón.

Haga doble clic en el control button1.

Visual Studio abre Form1.cs, que es "nuestra" parte del archivo de codificación, y se sitúa en el método button1_Click.

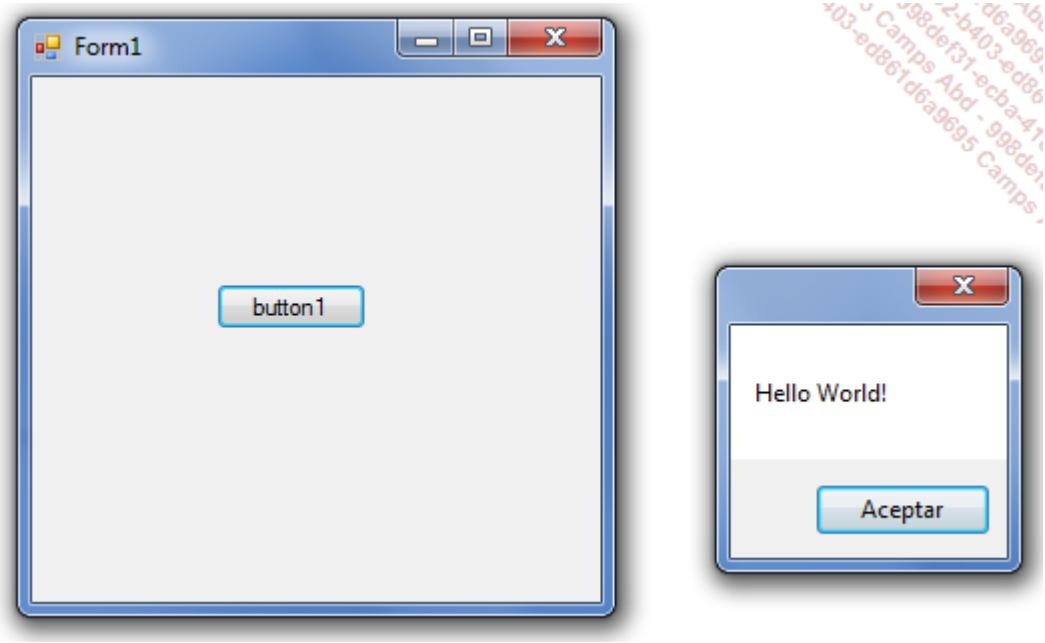
```
  private void button1_Click(object sender, EventArgs e)
  {


  }
```

Inserte en el método el siguiente comando:

```
MessageBox.Show("Hello World!");
```

Compile y ejecute la aplicación.

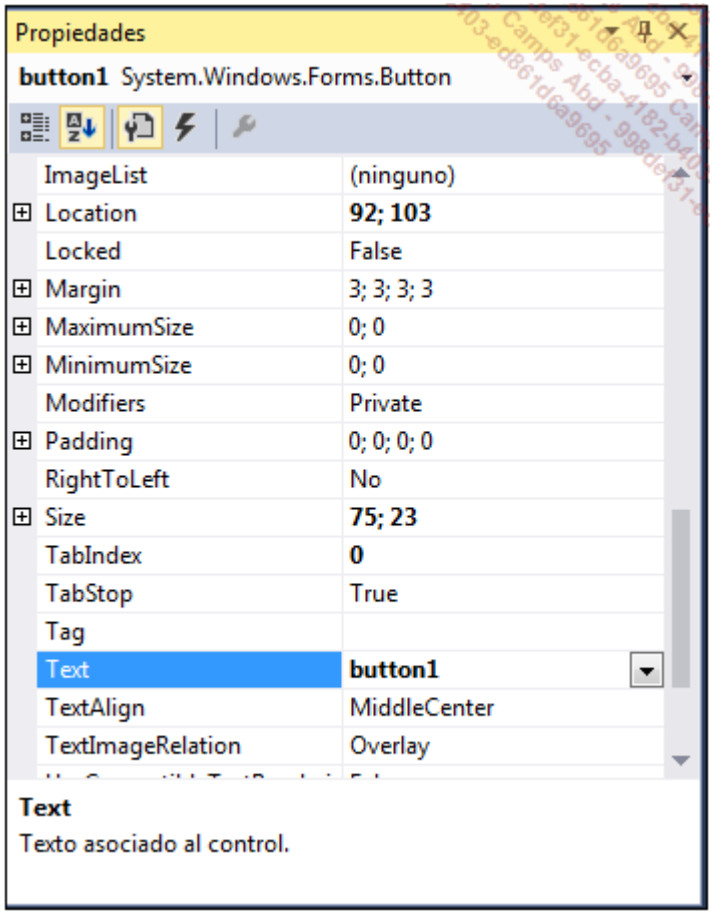
Pulse en button1 para comprobar el funcionamiento.



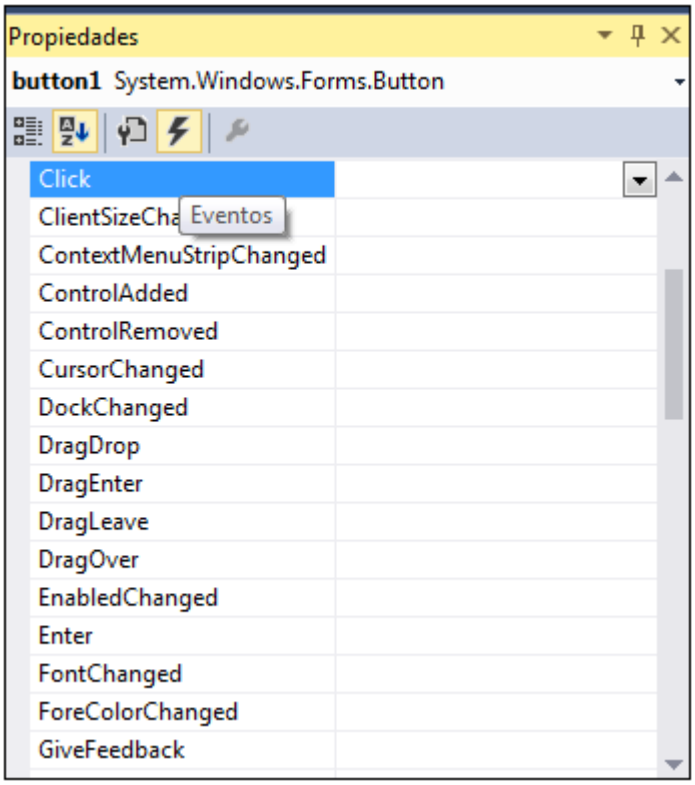
Pulse en **OK** y después cierre el formulario.

Veamos ahora cómo se realiza la unión.

En primer lugar, un control gráfico como el control button dispone de un determinado número de propiedades y event. Un clic con el botón derecho del ratón en **Propiedades** de button1 en el editor gráfico permite abrir el panel de propiedades en la parte inferior derecha.



Acceda a los event haciendo clic en el icono que representa un relámpago, en el cuadro de botones.



Click es un evento de un delegate de tipo EventHandler. Click forma parte de la clase Control, de la que hereda Button.

```
public class Control:...
{
  ...
  public event EventHandler Click;
}
```

Cuando se ha hecho doble clic en button1, el asistente ha suscrito la clase Form1 al evento Click de button1.

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

El método button1_Click corresponde al formato de un EventHandler "sencillo", es decir, con un EventArgs como segundo argumento.

```
private void button1_Click(object sender, EventArgs e){...}
```

El código del suscriptor también se llama "gestor de eventos". Un gestor de eventos en C#, al contrario de lo que sucede en la programación Java, no necesita implementar ninguna interfaz particular.

No hay eliminación de la suscripción al evento Click, porque button1 se crea por Form1 y su referencia es de tipo private en el objeto Form1. Por tanto, el botón está fuertemente unido a la clase Form1 y seguirá su ciclo de vida.

El asistente no puede utilizar una sintaxis basada en una expresión lambda porque el código de la clase Form1 se desarrolla en dos archivos de código fuente.

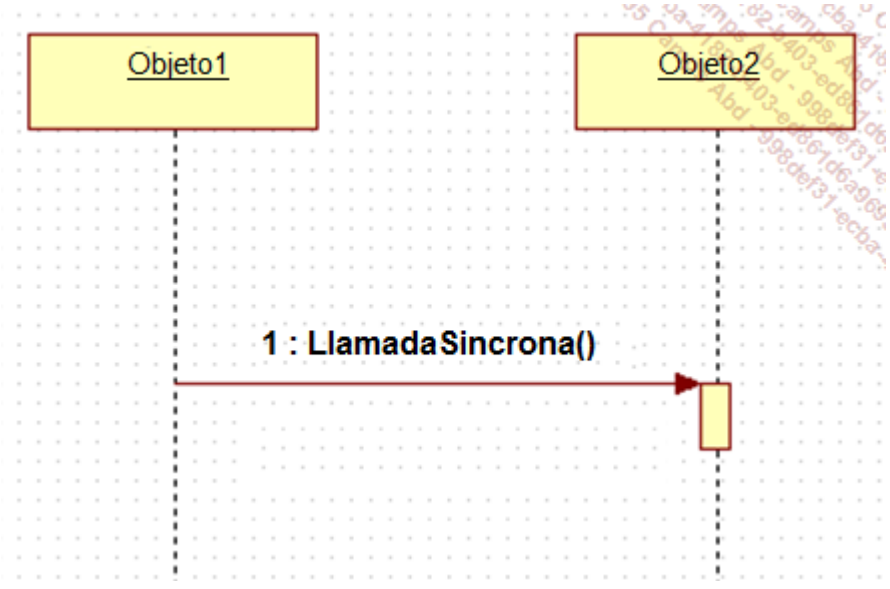
La comunicación por delegate se utiliza mucho en .NET.

Llamadas síncronas, llamadas asíncronas

Una noción importante afecta al modo de ejecución de un método respecto al programa que lo utiliza. De hecho, determinados métodos pueden necesitar bastante tiempo para ejecutarse. Por supuesto, esta noción de tiempo es subjetiva y depende del contexto de uso. Por ejemplo, en un entorno de oficina, un tiempo de ejecución de medio segundo no se representativo; en el dominio industrial no es lo mismo en absoluto.

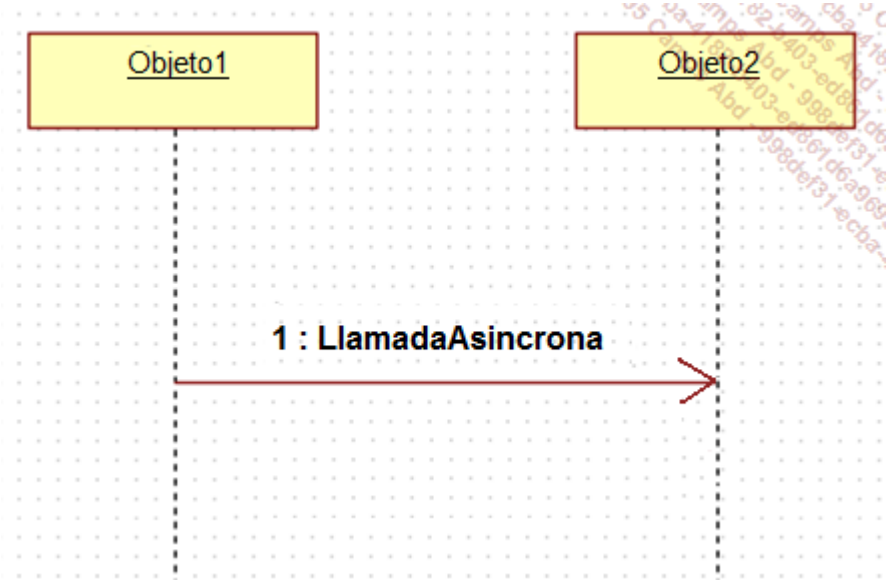
El funcionamiento síncrono es el modo de ejecución por defecto. El programa llamador permanece "bloqueado" durante la ejecución del método llamado.

A continuación se muestra la representación UML en forma de diagrama, de la secuencia de una llamada síncrona de una instancia Objeto1 a una instancia Objeto2.



Si la duración de la ejecución se convierte en crítica, hay que ejecutar el método de manera asíncrona. En este caso, la ejecución se "divide en dos", permitiendo que las dos ramas evolucionen en un modo "pseudo-paralelo".

A continuación se muestra la representación UML en forma de diagrama de secuencia de una llamada asíncrona de una instancia Objeto1 a una instancia Objeto2.



La programación de ejecuciones paralelas normalmente pasa por una programación multithread, que se presentará en el capítulo El multithreading. Sin embargo, los delegate ofrecen una alternativa sencilla para llamar a los métodos que tiene asociados de manera asíncrona, por lo que podemos aprovecharnos de ello.

En primer lugar, a continuación se muestra una clase que tiene un método que simula una ejecución durante un número definido de segundos gracias al método System.Threading.Thread.Sleep(xxx) que bloquea la ejecución durante un número definido de milisegundos. Este método es el que queremos ejecutar de manera asíncrona para la demonstración.

```
class MiClaseTrt
{
  public static string Job(int duracionEnSeg)
  {
    Thread.Sleep(duracionEnSeg * 1000);
    return "He trabajado " + duracionEnSeg + " segundos";
  }
}
```

A continuación se muestra la definición del delegate que lo encapsula:

```
public delegate string DelegateJob(int duracionEnSeg);
```

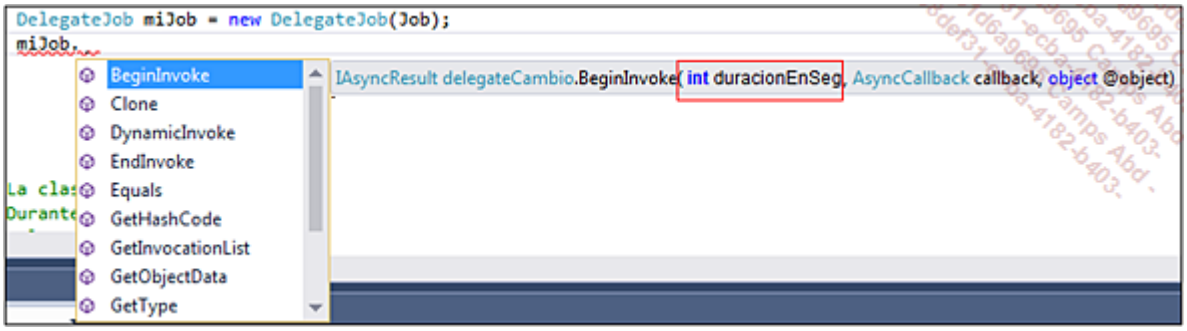
Y a continuación se muestra su instanciación:

```
DelegateJob miJob = new DelegateJob(MiClaseTrt.Job);
```

El tipo delegate ofrece:

- Un método Invoke() (o una llamada directa al delegate) para una invocación síncrona de sus suscriptores.
- Un método BeginInvoke() para una invocación asíncrona de sus suscriptores.

BeginInvoke() recibe los argumentos esperados por el método encapsulado. En nuestro caso, es solo uno: duracionEnSeg. Observe que IntelliSense ofrece el mismo nombre de variable.



A continuación, recibe la referencia a un objeto AsyncCallBack, que se presenta más adelante y permite definir una función de llamada al final de la ejecución, y un último argumento que dejaremos a null.

BeginInvoke() inicia la invocación y devuelve un objeto que soporta la interfaz IAsyncResult. Este objeto se va a utilizar por el programa llamador para identificar y comunicarse con el método que se está ejecutando. Esta comunicación se puede realizar de tres maneras diferentes.

1. Enfoque 1

```
Console.WriteLine("Prepara la ejecución asíncrona");
DelegateJob miJob = new DelegateJob(MiClaseTrt.Job);
IAsyncResult asyncResult = miJob.BeginInvoke(10, null, null);
Console.WriteLine("Ejecución asíncrona actual.");
Console.WriteLine("Aquí podemos realizar otras operaciones");
string answer = miJob.EndInvoke(asyncResult);
Console.WriteLine(answer);
```

Es el enfoque más sencillo. El método BeginInvoke() se llama con el argumento duracionEnSeg informado, los dos otros están a null. La ejecución es inmediata y el programa memoriza su valor de retorno antes de pasar al resto de operaciones.

Cuando termina sus actividades anexas, el programa principal llama al método EndInvoke(), lo que significa que las dos ejecuciones se vuelven a sincronizar.

Si durante la llamada a EndInvoke() la ejecución de Job ha finalizado, el retorno de EndInvoke() es inmediato. Si este no es el caso, la ejecución principal se bloquea hasta el final de la operación secundaria. Es el principal defecto de este primer enfoque.

2. Enfoque 2

Veamos el código de este segundo método:

```
Console.WriteLine("Prepara la ejecución asíncrona");
DelegateJob miJob = new DelegateJob(MiClaseTrt.Job);
IAsyncResult asyncResult = miJob.BeginInvoke(10, null, null);
Console.WriteLine("Ejecución asíncrona actual.");
while (!asyncResult.IsCompleted)
{
  Console.WriteLine(
          "Aquí podemos realizar otras operaciones");
  Thread.Sleep(1000);
}
string answer = miJob.EndInvoke(asyncResult);
Console.WriteLine(answer);
```

Este segundo método evita el bloqueo final llamando a EndInvoke() solo cuando la operación secundaria realmente ha terminado. Para esto, el programa principal pregunta, cuando le parece, sobre la propiedad IsCompleted del objeto que implementa IAsyncResult. Este enfoque es mejor que el anterior, pero el programa principal también debe realizar periódicamente una operación y esto puede incluso mejorar gracias al segundo argumento: AsyncCallBack.

3. Enfoque 3

Mejor uso de un delegate en modo asíncrono:

```
using System;
using System.Threading;

namespace DemoDelegadoAsync
{
  public delegate string DelegateJob(int duracionEnSeg);

  class Program
  {
    static void Main(string[] args)
    {
      MiClasePrincipal mcp = new MiClasePrincipal();
      mcp.JobPrincipal();
    }
  }

  class MiClaseTrt
  {
    public static string Job(int duracionEnSeg)
    {
      Thread.Sleep(duracionEnSeg * 1000);
      return "He trabajado " + duracionEnSeg + " segundos";
    }
  }

  class MiClasePrincipal
  {
    private bool isDone = false;

    public void JobPrincipal()
```

```
    {
      Console.WriteLine(
        "Thread {0} prepara la ejecución asíncrona",
        Thread.CurrentThread.ManagedThreadId);
      DelegateJob miJob = new DelegateJob(MiClaseTrt.Job);
      IAsyncResult asyncResult
        = miJob.BeginInvoke(10,
                            new AsyncCallback(JobComplete),
                            null);

      while (!isDone)
      {
        Thread.Sleep(1000);
        Console.WriteLine("Ejecución actual");
      }
      string answer = miJob.EndInvoke(asyncResult);
      Console.WriteLine(answer);
    }

    void JobComplete(IAsyncResult asyncResult)
    {
      Console.WriteLine("Fin de Job invocado por el thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
      isDone = true;
    }
  }
}
```

Este método utiliza ahora el ante penúltimo argumento de BeginInvoke(): AsyncCallback. Este argumento es una referencia a un delegate que encapsula un método de "llamada" entre la operación secundaria y la operación principal para indicar el final del trabajo.

Ahora, el programa principal tiene que comprobar únicamente una propiedad derivada de la operación secundaria; es la operación secundaria a la que llama cuando termina su trabajo.

Con el objetivo de simplificar el código de demostración, el método invocado actualiza un booleano que se comprueba por el programa que lo invoca para proseguir su trabajo.

Se asocia una unidad de ejecución a lo que se llama thread, que se presentará en el siguiente capítulo. En este fragmento de código, el thread principal lanza una operación asíncrona que se ejecuta en un thread secundario, como se puede comprobar en la siguiente salida por la consola.

```
Thread 1 prepara la ejecución asíncrona
Ejecución actual
Ejecución actual
Ejecución actual
Ejecución actual
Ejecución actual
Ejecución actual
Ejecución actual
Ejecución actual
Ejecución actual
Ejecución actual
Fin de Job invocado por el thread 3.
Ejecución actual....
He trabajado 10 segundos
Pulse una tecla para continuar...
```

Es importante entender que el método de llamada se va a ejecutar en un thread diferente al que ha lanzado la llamada asíncrona. Algunos componentes, como los objetos gráficos de un formulario, están muy asociados al thread que los ha creado y no se pueden modificar por otro thread. En el capítulo dedicado a la programación multithread veremos que la invocación es la solución a este problema.

4. Enfoque 3 con una expresión lambda

Como el método de llamada solo se va a ejecutar desde un único lugar, es posible utilizar una expresión lambda, reduciendo el tamaño del código.

```
using System;
using System.Threading;


namespace DemoDelegadoAsync
{
  public delegate string DelegateJob(int duracionEnSeg);
```

```
class Program
{
  static void Main(string[] args)
  {
    MiClasePrincipal mcp = new MiClasePrincipal();
    mcp.JobPrincipal();
  }
}

class MiClaseTrt
{
  public static string Job(int duracionEnSeg)
  {
    Thread.Sleep(duracionEnSeg * 1000);
    return "He trabajado " + duracionEnSeg + " segundos";
  }
}

class MiClasePrincipal
{
  public void JobPrincipal()
  {
    Console.WriteLine(
      "Thread {0} prepara la ejecución asíncrona",
      Thread.CurrentThread.ManagedThreadId);
    DelegateJob miJob = new DelegateJob(MiClaseTrt.Job);

    bool isDone = false;
    IAsyncResult asyncResult
      = miJob.BeginInvoke(
          10,
          (ar)=>
          {
```

```
              Console.WriteLine(
                 "Fin de Job invocado por el thread {0}.",
                 Thread.CurrentThread.ManagedThreadId);
              isDone = true;
            },
            null);

      while (!isDone)
      {
        Thread.Sleep(1000);
        Console.WriteLine("Ejecución actual");
      }
      string answer = miJob.EndInvoke(asyncResult);
      Console.WriteLine(answer);
    }

  }
}
```

La llamada a BeginInvoke() contiene el código de la expresión lambda. Como este código tiene acceso a las variables del método que lo invoca, puede actualizar el booleano, isDone que, al mismo tiempo, se convierte en una variable local.

De nuevo, esta parte del programa que realiza la llamada no está del todo optimizada.

```
while (!isDone)
      {
        Thread.Sleep(1000);
        Console.WriteLine("Ejecución actual");
      }
```

En el siguiente capítulo vamos a ver que hay objetos de sincronización que permiten evitar tener que hacer un bucle sobre una variable para saber cuándo termina el trabajo.

Ejercicio

El siguiente ejercicio va a permitir manipular los event, descubriendo una clase muy práctica de .NET: System.IO.File.FileSystemWatcher.

Esta clase tiene la particularidad de enviar notificaciones a sus suscriptores cuando se pasan eventos a los archivos en una carpeta específica de una unidad de almacenamiento.

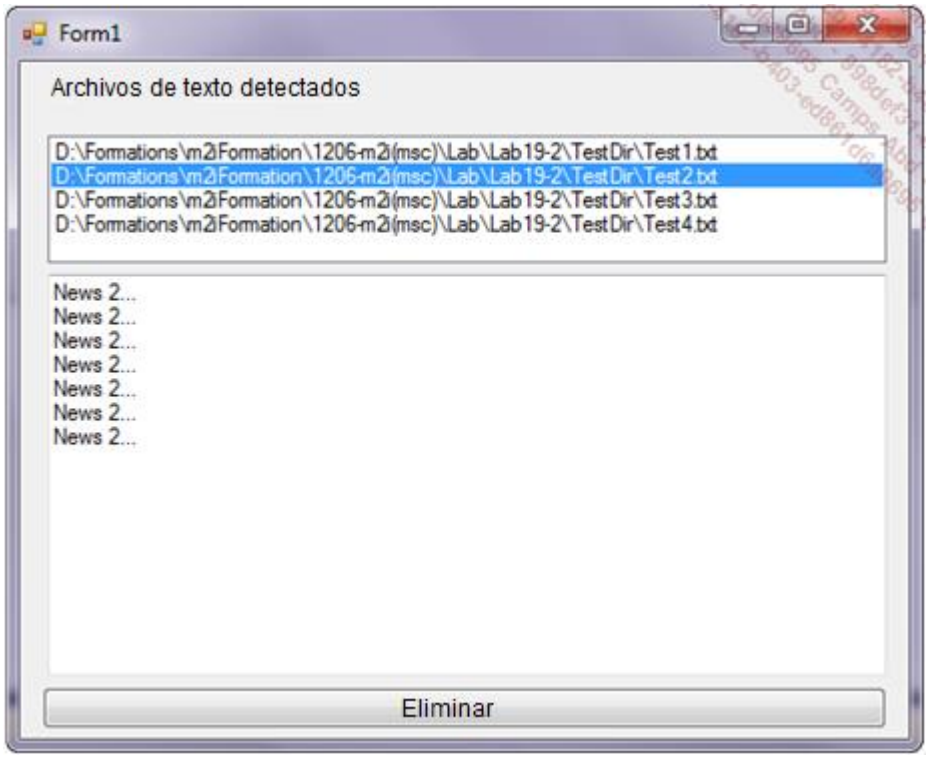
1. Enunciado

Vamos a utilizar FileSystemWatcher para monitorizar la escritura de archivos de extensión .TXT en una carpeta compartida, llamada por defecto C:\temp.

Se mostrará la lista de los archivos en un formulario y la selección de una entrada de la lista mostrará su contenido.

Una vez consultado el archivo, un botón **Eliminar** permitirá eliminar el archivo.

A continuación se muestra algo parecido al formulario:



2. Consejos para la realización

Crear un proyecto de tipo Windows Forms.

Crear un formulario en el asistente gráfico que contenga una listbox para mostrar los archivos detectados y una textbox multi-línea para mostrar el contenido de los archivos.

Utilice el evento Load del formulario para instanciar y configurar un objeto FileSystemWatcher.

Suscriba el formulario al evento Created del objeto FileSystemWatcher.

Tan pronto como se detecta un archivo, añada su nombre a la lista. Para evitar un problema que se comentará en el siguiente capítulo, utilizará:

```
this.listBox1.Invoke((MethodInvoker)delegate
                  {
                      this.listBox1.Items.Add(e.FullPath);
                  });
```

Suscriba el formulario al evento SelectedIndexChanged de la listbox para realizar la visualización del contenido del archivo.

Lea el archivo en un string con una sencilla llamada al método System.IO.File.ReadAllText.

Suscriba el formulario al evento Click del botón **Eliminar** y borre el archivo llamando al método System.IO.File.Delete.

3. Corrección

Parte del código de la clase generada por Visual Studio:

```
namespace LabFileSystemWatcher
{
  partial class Form1
  {
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.IContainer components = null;

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
```

```
    /// <param name="disposing">true if managed resources
    /// should be disposed; otherwise, false.</param>
    protected override void Dispose(bool disposing)
    {
      if (disposing && (components != null))
      {
          components.Dispose();
      }
      base.Dispose(disposing);
    }

    #region Windows Form Designer generated code

    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
      this.listBox1 = new System.Windows.Forms.ListBox();
      this.button1 = new System.Windows.Forms.Button();
      this.textBox1 = new System.Windows.Forms.TextBox();
      this.label1 = new System.Windows.Forms.Label();
      this.SuspendLayout();
      //
      // listBox1
      //
      this.listBox1.FormattingEnabled = true;
      this.listBox1.Location = new System.Drawing.Point(15, 40);
      this.listBox1.Name = "listBox1";
      this.listBox1.Size = new System.Drawing.Size(454, 69);
      this.listBox1.TabIndex = 0;
      this.listBox1.SelectedIndexChanged
```

```
            += new System.EventHandler(this.OnSelectionChanged);
        //
        // button1
        //
        this.button1.Location = new System.Drawing.Point(12, 338);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(457, 23);
        this.button1.TabIndex = 1;
        this.button1.Text = "Eliminar";
        this.button1.UseVisualStyleBackColor = true;
        this.button1.Click
            += new System.EventHandler(this.button1_Click);
        //
        // textBox1
        //
        this.textBox1.Location = new System.Drawing.Point(15, 115);
        this.textBox1.Multiline = true;
        this.textBox1.Name = "textBox1";
        this.textBox1.Size = new System.Drawing.Size(454, 217);
        this.textBox1.TabIndex = 2;
        //
        // label1
        //
        this.label1.AutoSize = true;
        this.label1.Location = new System.Drawing.Point(12, 9);
        this.label1.Name = "label1";
        this.label1.Size = new System.Drawing.Size(113, 13);
        this.label1.TabIndex = 3;
        this.label1.Text = "Archivos de texto detectados";
        //
        // Form1
        //
        this.AutoScaleDimensions
```

```
          = new System.Drawing.SizeF(6F, 13F);
      this.AutoScaleMode =
          system.Windows.Forms.AutoScaleMode.Font;
      this.ClientSize = new System.Drawing.Size(481, 366);
      this.Controls.Add(this.label1);
      this.Controls.Add(this.textBox1);
      this.Controls.Add(this.button1);
      this.Controls.Add(this.listBox1);
      this.Name = "Form1";
      this.Text = "Form1";
      this.Load += new System.EventHandler(this.OnLoading);
      this.ResumeLayout(false);
      this.PerformLayout();

    }

    #endregion

    private System.Windows.Forms.ListBox listBox1;
    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.TextBox textBox1;
    private System.Windows.Forms.Label label1;
  }
}
```

Parte del código de la clase implementada por el desarrollador:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace LabFileSystemWatcher
{
  public partial class Form1: Form
  {
    // Referencia a FileSystemWatcher
    private FileSystemWatcher fsw = null;

    public Form1()
    {
      InitializeComponent();
    }

    // Carga del formulario
    private void OnLoading(object sender, EventArgs e)
    {
      // Instanciación del FileSystemWatcher
      // con configuración de la carpeta a monitorizar
      fsw = new FileSystemWatcher(@"c:\temp")
      {
        // Se pone inmediatamente en funcionamiento
        EnableRaisingEvents = true
      };
```

```
    // Suscripción a su evento Created: el asistente de VS ha
    // creado el método "suscriptor" fsw_Created
    fsw.Created += new FileSystemEventHandler(fsw_Created);
  }

  // Método llamado por FileSystemWatcher durante la detección
  // de un nuevo archivo en la carpeta monitorizada.
  // Añade el nombre completo del archivo a la lista.
  void fsw_Created(object sender, FileSystemEventArgs e)
  {
    // fsw_Created se llama desde un thread
    // diferente del que se utiliza para crear el formulario.
    // La adición de la entrada en la lista se va a realizar
    // cuando el thread de creación se active
    // (Ver capítulo El multithreading)
    this.listBox1.Invoke((MethodInvoker)delegate
                {
                    this.listBox1.Items.Add(e.FullPath);
                });
  }

  // Método llamado cuando la selección en la lista cambia.
  // Permite actualizar la parte base del formulario
  // que muestra el contenido del archivo.
  private void OnSelectionChanged(object sender, EventArgs e)
  {
    int index = this.listBox1.SelectedIndex;
    if (index != -1)
    {
      string fullPath = this.listBox1.Items[index] as string;
      this.textBox1.Text =
                 System.IO.File.ReadAllText(fullPath);
    }
```

```
    }

    // Método llamado cuando el usuario pulsa en el botón
    // para eliminar el archivo.
    // La selección contiene el nombre del archivo completo.
    private void button1_Click(object sender, EventArgs e)
    {
      int index = this.listBox1.SelectedIndex;
      if (index != -1)
      {
        string fullPath = this.listBox1.Items[index] as string;
        System.IO.File.Delete(fullPath);
        this.listBox1.Items.RemoveAt(index);
        this.textBox1.Text = "";
      }
    }
  }
}
```

Una vez cargado el formulario, se arrastran los archivos de extensión TXT a la carpeta C:\temp, para comprobar el funcionamiento de la aplicación.

Introducción

La programación multithread es un dominio apasionante, pero que rápidamente se puede convertir en algo muy complejo de afinar. Varias ejecuciones paralelas en el núcleo de su aplicación deberán compartir información, esperarse e intercambiar datos. El éxito de una arquitectura de este tipo reposa sobre todo en un análisis sólido. Este capítulo no tiene la intención de explicar todas las posibilidades de programación multithread y sus implementaciones en .NET, sino presentar los aspectos fundamentales relacionados con la filosofía POO.

Entender el multithreading

Un programa puede realizar operaciones largas que van a "bloquear" la aplicación durante su ejecución. Para evitar esto, el desarrollador puede crear un tipo de ejecución en paralelo que va a ser responsable de esta operación y, de esta manera, liberar al ejecutor principal. En este caso, el sistema operativo Windows de Microsoft comparte muy rápidamente el tiempo de máquina entre los diferentes flujos de ejecución (típicamente, 20 ms por intervalo de tiempo), dando la impresión de una ejecución simultanea.

Hablamos de sistema operativo en tiempo compartido. El contenido de un hilo de ejecución puede encadenar todas las operaciones que quiera, sin preocuparse por el tiempo que esto llevará a nivel global. El sistema operativo lo interrumpirá periódicamente para dar tiempo al hilo de ejecución siguiente y, de esta manera, continuar hasta volver al primero para que retome su operación donde la dejó.

Tomemos como ejemplo el receptor de entradas/salidas, equipado con una interfaz de programación muy resumida. El constructor nos da un juego de funciones que permite, entre otras cosas, leer el estado binario de las entradas numeradas. Desea desarrollar una aplicación domótica que gestione varias operaciones en paralelo, como la iluminación, la calefacción e incluso la alarma de intrusión. Con lo que se presentó sobre las notificaciones usando event, desea transformar la interfaz básica de programación en un módulo observable sofisticado capaz de llamar a instancias de objetos suscriptores tan pronto como un sensor detecte un cambio de estado. Para ello, crea una operación dentro de un bucle dedicado a la lectura de cada entrada de la tarjeta. Gracias a algunas operaciones sabrá detectar un cambio real de estado y desencadenar el evento asociado. Las operaciones que contienen los gestores de eventos, llamadas operaciones de negocio porque son estas las que asignan los cambios de estado, no se "bloquean" en su ejecución. Pueden continuar mostrando información, imprimiendo históricos o realizando cálculos.

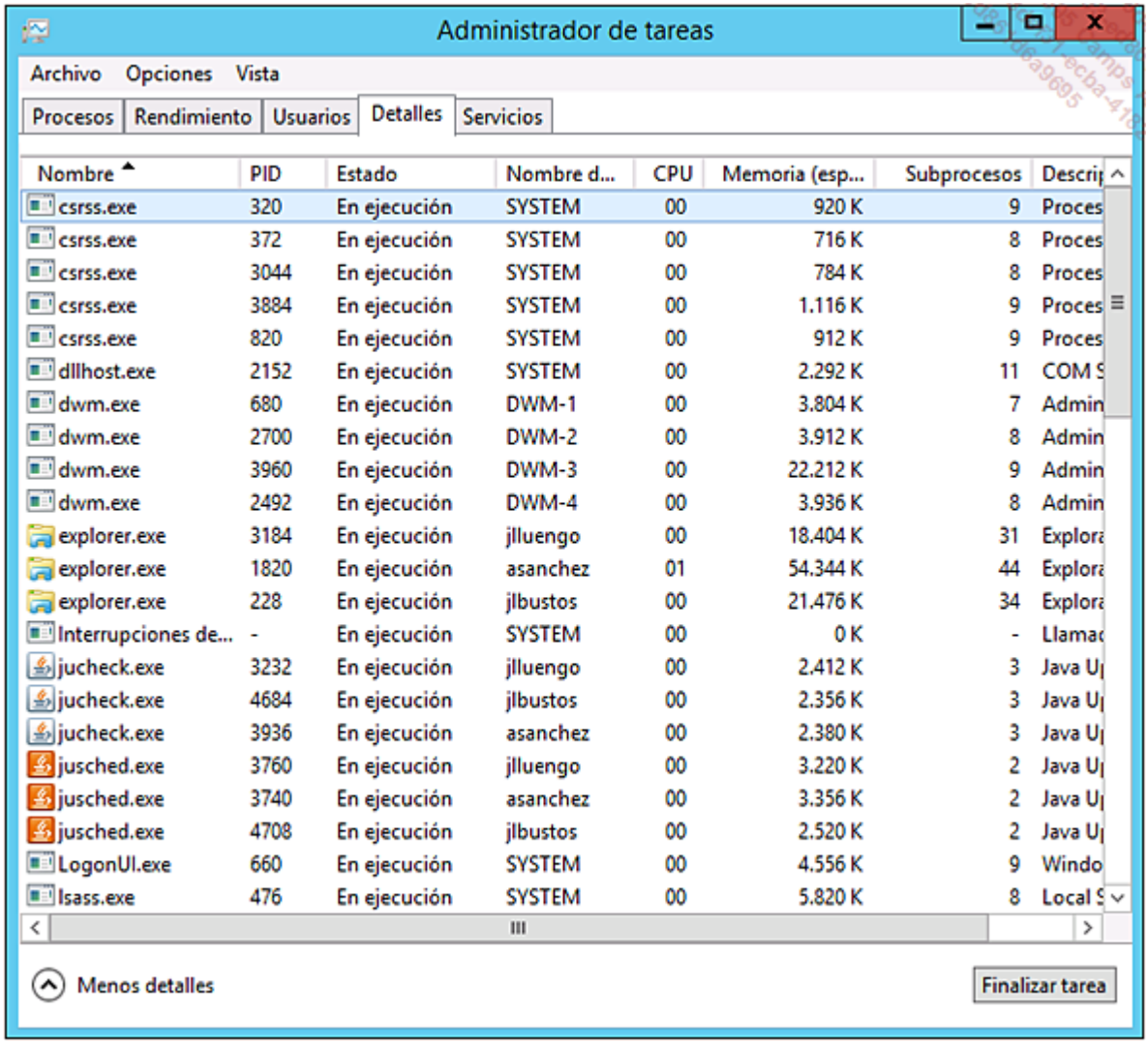
Estos flujos de ejecución se llaman threads. Un thread puede tener o no interacciones con otras partes de la aplicación. Si es el caso, siempre hay que pensar que la ejecución se puede interrumpir por el sistema operativo en cualquier momento, incluido durante la actualización de objetos que quedarán en un estado temporal hasta el siguiente ciclo. Si estos objetos se comparten sin cuidado con otras partes de la aplicación, habrá funcionamientos incorrectos provocados por estos cambios de contexto previamente explicitados.

Los threads tienen características de funcionamiento, entre las que figura la noción de prioridad. En el caso de una adquisición de un flujo de datos es importante no perder información. Si el sistema está muy ocupado, se corre el riesgo de volver a lanzar el thread de adquisición después de la saturación de la memoria RAM de datos del hardware y, por tanto, perder datos. Es posible intervenir sobre el funcionamiento del gestor de ejecuciones incrementando la prioridad del thread. Este cambio de prioridad solo se debe realizar en casos muy particulares que lo justifiquen.

Por último, es importante distinguir thread y proceso. Cuando el usuario ejecuta un archivo de extensión .EXE se crea un proceso. Los procesos son bloques de ejecución que contienen las asignaciones de memoria y recursos necesarios. Los procesos están aislados y, por tanto, no se pueden "invadir" los unos a los otros. La única manera de que se puedan comunicar dos procesos entre ellos es pasarlos por una IPC (Interprocess Communication) basada en pipes, mailslots o sockets.

.NET ofrece además una API muy notable, llamada WCF (Windows Communication Foundation), que permite la ejecución de aplicaciones de software distribuidas en una o varias máquinas. A su vez, pueden estar en ubicaciones geográficas diferentes.

Cada proceso se identifica por su PID (Process Identifier) y contiene al menos un thread, llamado "subproceso primario", que se crea automáticamente. A continuación, los threads secundarios se pueden añadir por programación. El administrador de tareas de Windows permite mostrar la lista de procesos activos y también conocer el número de subprocesos de cada uno.



Si las columnas **PID** y **Subprocesos** no se muestran, hay que seleccionarlas desde el menú contextual de la tabla que se muestra haciendo clic con el botón derecho del ratón en la cabecera de las columnas.

Los threads de un mismo proceso son internos a este proceso, así como a un mismo espacio de memoria, un mismo código y los mismos recursos. Cada thread dispone de su propio stack y también de una zona de memoria (TLS, de Thread Local Storage) utilizada para guardar automáticamente los datos durante su periodo de inactividad.

Los procesos cargan módulos que están en forma de archivos DLL. El ciclo de vida de estos módulos está totalmente vinculado al de los procesos.

Multithreading y .NET

La encapsulación de gestión de procesos se proporciona a través del tipo System.Diagnostics.Process.

El siguiente fragmento de código permite ejecutar el programa Windows calc.exe desde un programa .NET utilizando los tipos Process y ProcessStartInfo.

```
using System;
using System.Diagnostics;


namespace DemoAppDomain
{
  class Program
  {
    static void Main(string[] args)
    {
      ProcessStartInfo psi = new ProcessStartInfo("calc.exe");
      Process.Start(psi);


      Console.ReadKey();
    }
  }
}
```

La clase Process permite realizar otras acciones, como la enumeración de los procesos activos y la lectura de sus características: PID, nombre y número de threads.

Existe una capa intermedia entre el proceso del sistema operativo y su aplicación .NET gestionada. Esta capa se llama dominio de aplicación (AppDomain). Cuando se ejecuta la aplicación, la CLR genera un AppDomain por defecto, que establece la relación con el proceso real del sistema operativo. La mayor parte del tiempo hay tantos procesos como aplicaciones .NET ejecutándose y el desarrollador no tiene que hacer nada particular. Sin embargo, en el caso de aplicaciones complejas que contienen varios módulos, es posible optimizar el conjunto cargando varios ensamblados en un único dominio de aplicación. Este enfoque es mucho menos costoso en términos de recursos que el uso de un proceso del sistema. También es posible descargar un dominio de aplicación que no es útil.

El desarrollador puede asegurarse de que la memoria ocupada por el conjunto de los ensamblados se libere sin retraso. La clase System.AppDomain permite programar los dominios de aplicaciones.

El espacio de nombres System.Threading proporciona clases e interfaces para programar en multithreading. Este espacio de nombres ofrece al desarrollador las herramientas para crear threads, así con un juego de threads "listos para usar" (ThreadPool). También contiene una clase Timer que permite llamar periódicamente a un delegate.

Existen dos tipos de threads en .NET:

- El thread de tipo Foreground: la aplicación que lo ha creado no se puede cerrar mientras que esté activo.
- El thread de tipo Background: la aplicación que lo ha creado se puede cerrar mientras trabaja, provocando su parada inmediata.

Implementación en C#

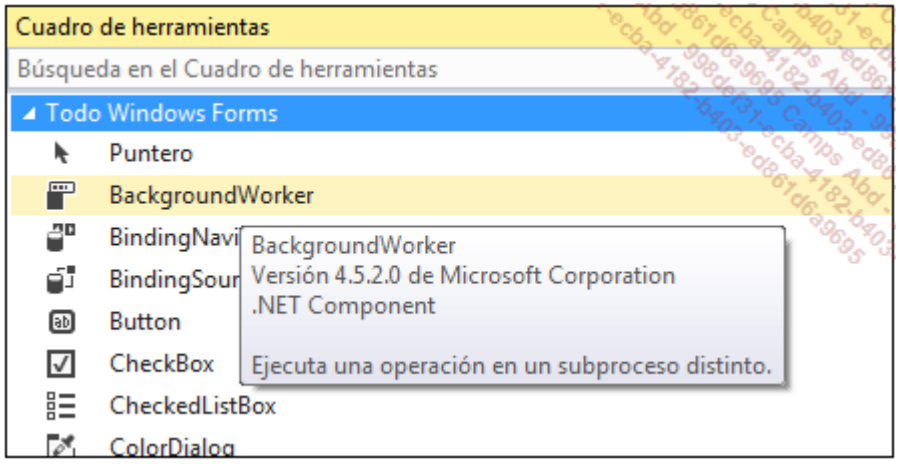
Existen tres maneras principales de programar los threads en C#.

1. Uso de un BackgroundWorker

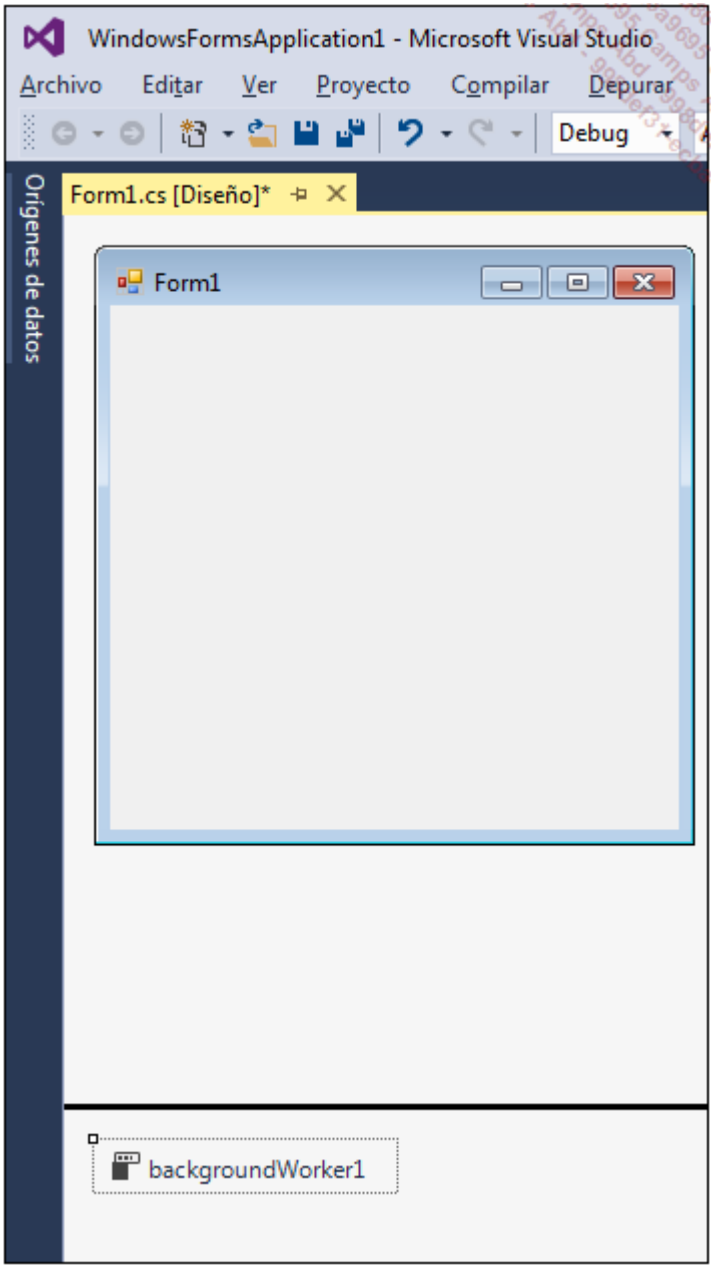
La primera programación de thread consiste en utilizar el editor de recursos de Visual Studio para añadir a un formulario un objeto de tipo System.ComponentModel.BackgroundWorker.

Atención, un thread BackgroundWorker siempre será de tipo background y siempre tendrá una prioridad "normal".

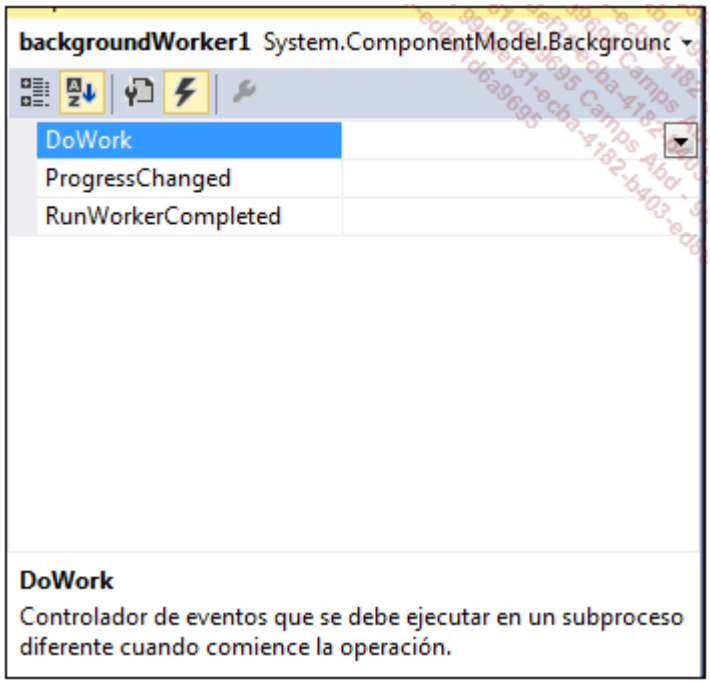
Para crear este tipo de threads de la manera más sencilla posible basta con desplegar el **Cuadro de herramientas**, seleccionar el BackgroundWorker de la lista de componentes y arrastrarlo al formulario que se está desarrollando.



Naturalmente, el componente BackgroundWorker no encapsula un control gráfico, aparece debajo de la zona de visualización.



La codificación de la instanciación del objeto BackgroundWorker se lleva a cabo por el asistente gráfico y el dato miembro toma el nombre por defecto de backgroundWorker1. Ahora, es suficiente con conectar al componente los métodos del formulario que se van a llamar durante la ejecución del thread. Esto se realiza desde la ventana **Propiedades** de backgroundWorker1...



Por tanto, la explotación del thread en el formulario pasa por los tres eventos disponibles en el objeto backgroundWorker1:

- DoWork: apunta al método que se debe ejecutar por el thread.
- ProcessChanged: apunta a un método que se va a llamar durante la ejecución del trabajo para mostrar la información del avance. Este método puede modificar los componentes del formulario.
- RunWorkerCompleted: apunta al método que se llama cuando el trabajo termina. También puede modificar la visualización.

Estos tres eventos funcionan con delegate adaptados. El nombre de los delegate corresponde al nombre del evento seguido de EventHandler. Por ejemplo, DoWorkEventHandler es el delegate de DoWork.

Una vez más, el asistente de Visual Studio se ocupa de preparar todo, pero es necesario que conozcamos las posibilidades de comunicación entre la aplicación (el thread principal) y el thread secundario.

a. Comunicación del thread principal con el thread secundario

Durante su ejecución, el thread secundario recibe como argumento un objeto de tipo DoWorkEventArgs, como se puede comprobar en la firma de su delegate DoWork EventHandler.

```
public delegate void DoWorkEventHandler(object sender,
                                        DoWorkEventArgs e);
```

DoWorkEventArgs recibe un objeto de cualquier tipo en su constructor que expone a continuación en modo solo lectura.

El thread principal transmite (o puede transmitir) este objeto cuando inicia el thread durante la llamada de RunWorkerAsync.

```
int maxi = 5;
myBackgroundWorker.RunWorkerAsync(maxi);
```

El thread secundario recupera después la información cuando arranca.

```
       // Gestor de eventos de inicio del trabajo
        void myBackgroundWorker_DoWork(object sender,
                                       DoWorkEventArgs e)
        {
            // Recuperación del argumento que se pasa
            int max = (int)e.Argument;
```

b. Abandono desde el thread principal

Durante la ejecución del thread secundario, el usuario puede decidir interrumpir la operación. Para esto, el objeto backgroundWorker1 ofrece un método CancelAsync que actualiza un booleano que el thread secundario puede comprobar durante su ejecución. Este booleano se llama CancellationPending. Algunas líneas más abajo hay un ejemplo que muestra el uso de este mecanismo.

c. Comunicación del thread secundario con el thread principal

Durante su ejecución, el thread secundario puede llamar periódicamente a un método del objeto BackgroundWorker para informar al thread principal del avance de su trabajo. En el siguiente código, la variable i representa el porcentaje realizado, pero el método ReportProgress se sobrecarga y puede recibir un segundo argumento de tipo Object (o derivado).

```
myBackgroundWorker.ReportProgress(i);
```

A continuación se muestra el prototipo del delegate, que recibe las notificaciones del thread secundario.

```
public delegate void ProgressChangedEventHandler(object sender,
                                    ProgressChangedEventArgs e);
```

El objeto ProgressChangedEventArgs expone una propiedad ProgressPercentage que se actualiza durante la llamada a ReportProgress. Por tanto, el thread principal puede utilizarla para actualizar un control gráfico como, por ejemplo, una barra de progreso.

d. Comunicación al final de la operación del thread secundario

Cuando arranca, el thread secundario recibe un objeto de tipo DoWorkEventArgs. Este es el objeto que también va a servir de enlace para navegar entre el thread secundario y el principal, esta vez a través de su miembro Result, que es de tipo Object, por tanto cualquiera. De esta manera, el thread secundario puede transmitir el resultado de su trabajo al thread principal.

e. Ejemplo de código

En el siguiente ejemplo, el formulario crea un control de tipo barra de progreso y un thread secundario que simula un trabajo "largo" en una tabla (myBackgroundWorker_DoWork). En cada nueva lectura de la tabla, el thread secundario informa al formulario de su avance (myBackgroundWorker.ReportProgress(i)). Para terminar, cuando su trabajo termina, el thread secundario transmite la información "Thread terminado" al formulario.

```
using System;
using System.ComponentModel;
using System.Threading;
using System.Windows.Forms;

namespace PruebaThreadBackgroundWorker
{
    public partial class Form1: Form
    {
        const int maxi = 5;

        private BackgroundWorker myBackgroundWorker;
        public Form1()
        {
            InitializeComponent();
            this.myBackgroundWorker
              = new System.ComponentModel.BackgroundWorker();
            // Indica que el thread podrá notificar
            // su evolución (event ProgressChanged)
            myBackgroundWorker.WorkerReportsProgress = true;
            // Indica que el thread podrá ser
            // suscriptor (CancelAsync)
            myBackgroundWorker.WorkerSupportsCancellación = true;
            // Init del gestor de eventos de
            // lanzamiento del trabajo
            myBackgroundWorker.DoWork += new
                  DoWorkEventHandler(myBackgroundWorker_DoWork);
            // Init del gestor de eventos de
            // progreso del thread
            myBackgroundWorker.ProgressChanged += new
            ProgressChangedEventHandler(
                            myBackgroundWorker_ProgressChanged);
            // Init del gestor de eventos de fin del trabajo
```

```
            myBackgroundWorker.RunWorkerCompleted += new
            RunWorkerCompletedEventHandler(
                          myBackgroundWorker_RunWorkerCompleted);
            // Inicio del thread con transmisión
            // de argumento: aquí un int
            myBackgroundWorker.RunWorkerAsync(maxi);

            myProgressBar.Minimum = 0;
            myProgressBar.Maximum = maxi;
            labelStatus.Text = "Thread actual";
        }

        // Gestor de eventos de inicio del trabajo
        void myBackgroundWorker_DoWork(object sender,
                                       DoWorkEventArgs e)
        {
            System.Diagnostics.Debug.WriteLine(
              "IsBackground: "
              +Thread.CurrentThread.IsBackground);

            // Recuperación del argumento que se pasa como
            // RunWorkerAsync
            int max = (int)e.Argument;
            for (int i = 0;
            i <= max  && !myBackgroundWorker.CancellationPending;
            i++)
            {
                // Llamada del gestor de eventos
                // progreso del thread
                myBackgroundWorker.ReportProgress(i);
                Thread.Sleep(1000);
            }
            e.Result = "Thread terminado ";
```

```
        }

        // Gestor de eventos de progreso del thread
        void myBackgroundWorker_ProgressChanged(object sender,
                                    ProgressChangedEventArgs e)
        {
            myProgressBar.Value = e.ProgressPercentage;
        }

        // Gestor de eventos
        // de fin del trabajo del thread
        void myBackgroundWorker_RunWorkerCompleted(object sender,
                                   RunWorkerCompletedEventArgs e)
        {
            labelStatus.Text = e.Result as string;
            buttonCancel.Enabled = false;
        }

        // Botón de abandono del thread
        private void buttonCancel_Click(object sender,
                                        EventArgs e)
        {
            if (myBackgroundWorker.IsBusy)
            {
                labelStatus.Text = "Thread en proceso de abandono ";
                myBackgroundWorker.CancelAsync();
            }
        }
    }
}
```

2. Utilización del pool de threads creado por .NET

En el capítulo anterior hemos visto que un delegate podía funcionar en modo síncrono y asíncrono. Este modo de funcionamiento asíncrono se basa naturalmente en el uso de threads pertenecientes a una colección "lista para usar", que la CLR mantiene por nosotros. Tan pronto como el trabajo termina, el thread vuelve a la fila de threads disponibles.

La buena noticia es que también podemos aprovechar este juego disponible y utilizar los threads sin necesitar recursos adicionales. La mala noticia es que los threads a nuestra disposición siempre serán de tipo background y de prioridad normal. Por otra parte, los días de gran actividad, habrá que esperar su turno.

Para utilizar la colección, el espacio de nombres System.Threading ofrece la clase ThreadPool y su método de tipo static QueueUserWorkItem. Este último permite ejecutar código de manera totalmente asíncrona, pasándole eventualmente un argumento de tipo Object (por tanto, cualquiera).

A continuación se muestra un ejemplo de código que utiliza System.Threading.ThreadPool.

La clase Prueba contiene un método principal que lanza una operación en paralelo pasándole un argumento. La operación secundaria simula un trabajo con una duración proporcional al valor que recibe.

```
using System;
using System.Threading;



namespace PruebaThread
{
  class Program
  {
    static void Main(string[] args)
    {
      Prueba t = new Prueba();
      t.TratamientoPrincipal();
      Console.ReadLine();
    }

    class Prueba
    {
      public void TratamientoPrincipal()
```

```
      {
        Console.WriteLine("Inicio TratamientoPrincipal");
        ThreadPool.QueueUserWorkItem(
                          new WaitCallback(TratamientoSecundario)
                          ,10);
        Thread.Sleep(1000 * (10+1));
        Console.WriteLine("Fin TratamientoPrincipal");
      }

      private void TratamientoSecundario(object o)
      {
        Console.WriteLine("Inicio TratamientoSecundario");
        Thread.Sleep(1000*(int)o);
        Console.WriteLine("Fin TratamientoSecundario");
      }
    }
  }
}
```

Salida por consola asociada:

```
Inicio TratamientoPrincipal
Inicio TratamientoSecundario
Fin TratamientoSecundario
Fin TratamientoPrincipal

Pulse una tecla para continuar...
```

Dando por hecho que TratamientoSecundario solo se llama por TratamientoPrincipal, es posible "simplificar" el código utilizando una expresión lambda.

```
using System;
using System.Threading;

namespace PruebaThread
{
  class Program
  {
    static void Main(string[] args)
    {
      Prueba t = new Prueba();
      t.TratamientoPrincipal();
      Console.ReadLine();
    }

    class Prueba
    {
      public void TratamientoPrincipal()
      {
        Console.WriteLine("Inicio TratamientoPrincipal");
        ThreadPool.QueueUserWorkItem(
          tempo =>
          {
            Console.WriteLine("Inicio TratamientoSecundario");
            Thread.Sleep(1000*(int)tempo);
            Console.WriteLine("Fin TratamientoSecundario");
          }
          ,10);
        Thread.Sleep(1000 * (10+1));
        Console.WriteLine("Fin TratamientoPrincipal");
      }
    }
  }
}
```

ThreadPool es muy sencillo de implementar y está optimizado, porque los threads se crean y se mantienen por la CLR. Su número depende del hardware utilizado. Elegirá esta solución si no necesita cambiar el tipo, prioridad o identificar a sus threads.

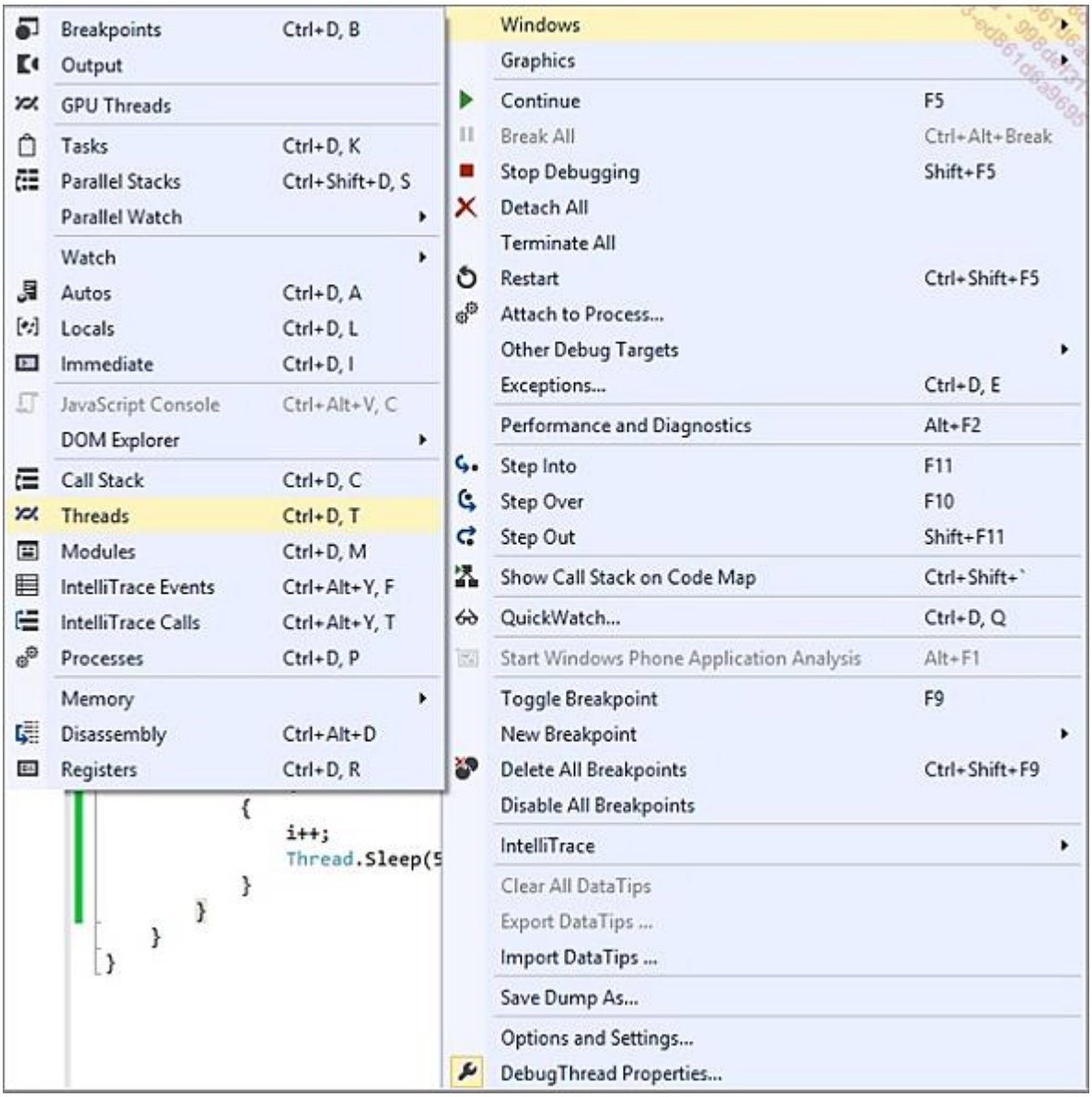
3. Gestión «manual» con Thread/ParameterizedThreadStart

La gestión manual del thread sigue siendo la opción más compleja, pero la más potente, porque permite:

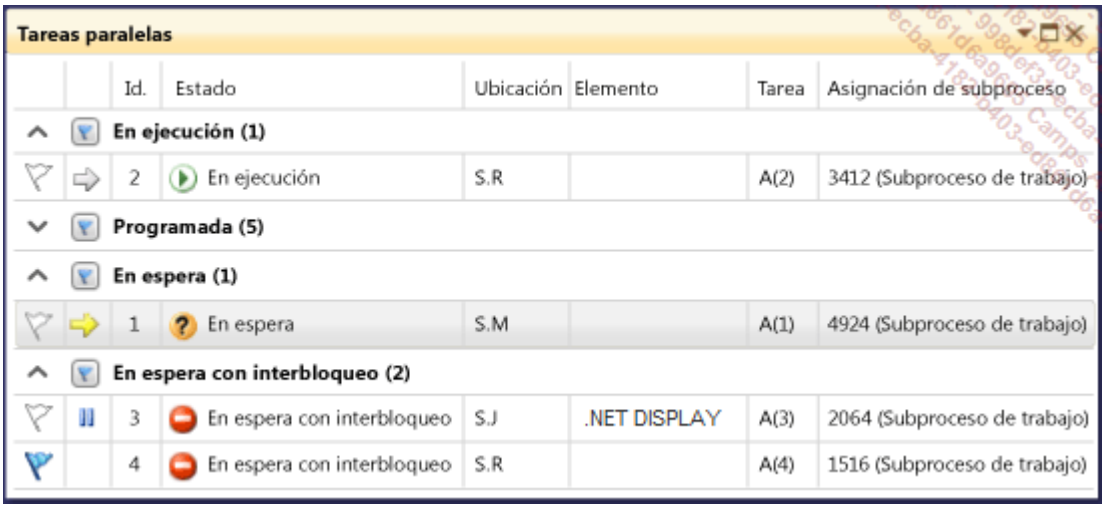
- ajustar el tipo del thread: Foreground o Background.
- ajustar su nivel de prioridad Highest, AboveNormal, Normal, BelowNormal o Lowest.
- definir su nombre.

Dar nombre a un thread resulta particularmente interesante durante las fases de puesta a punto de su programa, porque Visual Studio ofrece cuadros de diálogo que permiten mostrar los threads y un nombre explícito siempre es más sencillo de utilizar que un identificador.

Petición de visualización de la lista de threads:



Ejemplo de lista de threads. El último es un thread "con nombre":



A continuación se describe la secuencia a seguir cuando se desee programar un thread manualmente:

Crear un método "trabajo", al cual va a llamar el thread.

Si no tiene que enviar argumentos a su thread, cree un delegate de tipo ThreadStart que reciba el método "trabajo" creado anteriormente como argumento de su constructor.

Si tiene que enviar un argumento a su thread, cree un delegate de tipo ParameterizedThreadStart que también reciba el método "trabajo" como argumento de su constructor.

Eventualmente, puede configurar el tipo, la prioridad y el nombre del thread.

Invoque a Thread.Start() si el thread no recibe argumentos.

Invoque a Thread.Start(miObjetoDeConfig) si el thread espera recibir un objeto de configuración.

A continuación se muestra un ejemplo de código que no pasa argumentos entre el thread principal y el thread secundario:

```
using System;
using System.Threading;

namespace PruebaThread
{
  class Program
  {
    static void Main(string[] args)
    {
      Prueba t = new Prueba();
      t.TratamientoPrincipal();
      Console.WriteLine(
        "Pulse una tecla para salir");
    }


    class Prueba
    {
      public void TratamientoPrincipal()
      {
        Console.WriteLine("Inicio TratamientoPrincipal");
        ThreadStart ts = new ThreadStart(TratamientoSecundario);
        Thread t = new Thread(ts);
        t.IsBackground = false;
        t.Priority = ThreadPriority.Highest;
        t.Name = "Es mi thread :)";
        t.Start();

        Console.WriteLine("Fin TratamientoPrincipal");
      }

      private void TratamientoSecundario()
```

```
      {
        Console.WriteLine("Inicio TratamientoSecundario");
        Thread.Sleep(1000 * 10);
        Console.WriteLine("Fin TratamientoSecundario");
      }
    }
  }
}
```

Si ejecuta este código y pulsa una tecla antes del final del thread secundario comprobará que la aplicación está bloqueada mientras el trabajo termina. Es normal, el thread creado es de tipo foreground.

A continuación se muestra un ejemplo de código que pasa argumentos entre el thread principal y el thread secundario:

```
using System;
using System.Threading;


namespace PruebaThread
{
  class Program
  {
    static void Main(string[] args)
    {
      Prueba t = new Prueba();
      t.TratamientoPrincipal();
      Console.WriteLine(
        "Pulse una tecla para salir");
    }



    class Prueba
    {
      public void TratamientoPrincipal()
      {
        Console.WriteLine("Inicio TratamientoPrincipal");
```

```
        ParameterizedThreadStart ts
          = new ParameterizedThreadStart(TratamientoSecundario);
        Thread t = new Thread(ts);
        t.IsBackground = false;
        t.Priority = ThreadPriority.Highest;
        t.Name = "Es mi thread :)";
        t.Start(10);

        Console.WriteLine("Fin TratamientoPrincipal");
      }

      private void TratamientoSecundario(object o)
      {
        Console.WriteLine("Inicio TratamientoSecundario");
        Thread.Sleep(1000 * (int)o);
        Console.WriteLine("Fin TratamientoSecundario");
      }
    }
  }
}
```

La sintaxis actual es un poco "pesada". Comenzando por que TratamientoSecundario solo se llama por TratamientoPrincipal, es aconsejable una optimización con expresión lambda.

```
using System;
using System.Threading;

namespace PruebaThread
{
  class Program
  {
    static void Main(string[] args)
    {
      Prueba t = new Prueba();
      t.TratamientoPrincipal();
```

```
      Console.WriteLine(
        "Pulse una tecla para salir");
    }



    class Prueba
    {
      public void TratamientoPrincipal()
      {
        Console.WriteLine("Inicio TratamientoPrincipal");
        int numSec = 10;
        Thread t = new Thread(
          new ThreadStart(() =>
              {
                Console.WriteLine("Inicio TratamientoSecundario");
                Thread.Sleep(1000 * numSec);
                Console.WriteLine("Fin TratamientoSecundario");
              }
            ));
        t.IsBackground = false;
        t.Priority = ThreadPriority.Highest;
        t.Name = "Es mi thread :)";
        t.Start();

        Console.WriteLine("Fin TratamientoPrincipal");
      }
    }
  }
}
```

En este fragmento de código, el objeto ParameterizedThreadStart se ha podido sustituir por un objeto de tipo ThreadStart porque la expresión lambda tiene acceso a las variables locales del método que la alberga y, por tanto, tiene acceso a numSec.

Sincronización entre threads

1. Necesidad de la sincronización

La programación de varias rutas de ejecución no plantea ningún problema particular, hasta que comparten la misma información o los mismos recursos. Dando por hecho que el sistema operativo puede interrumpir las operaciones en cualquier momento, se corre el riesgo de tener objetos que se están modificando en un thread prioritario y que se encuentre en algún estado inestable para el thread siguiente. Para prevenir estos funcionamientos incorrectos hay que "sincronizar" los threads, es decir, proteger las zonas de operación delicadas.

Esto no lo va a realizar el sistema de gestión, que continuará activando los threads unos después de otros; sencillamente, cuando un thread A necesite acceder a una dato común protegido porque un thread B no ha terminado de actualizarlo, el thread A deberá "esperar a la siguiente vuelta". Y si durante la siguiente vuelta el trabajo del thread B no ha terminado todavía, deberá esperar a la siguiente, y así sucesivamente.

El principio es el mismo si se trata de una operación común, de modo que el thread B deberá haber terminado antes de que el thread A lo pueda realizar en su turno. Este es el escenario que muestra el siguiente fragmento de código. De hecho, la operación permite mostrar una cuenta desde cero hasta nueve, realizada por diez threads. El objetivo esperado es la siguiente visualización:

```
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
Pulse una tecla para continuar...
```

A continuación se muestra una primera version del código "sin protección":

```
using System;
using System.Threading;
```

```
namespace PruebaThreadSinSincro
{
  class Program
  {
    static void Main(string[] args)
    {
      Prueba t = new Prueba();
      t.TratamientoPrincipal();
    }
  }
  class Prueba
  {
    public void TratamientoPrincipal()
    {
      for (int i = 0; i < 10; i++)
      {
        new Thread(new ThreadStart(
          () =>
          {
            for (int j = 0; j < 10; j++)
            {
              Console.Write(j);
              Thread.Sleep(0);
            }
            Console.WriteLine();
          }
          )).Start();
      }
    }
  }
}
```

Nada más que lo que se ha presentado: un bucle que inicia diez threads que muestran cada uno una cuenta que va de cero hasta nueve. Observe una llamada a Thread.Sleep(0) en el bucle de visualización. Concretamente, la llamada a esta función permite decir al ejecutor del

thread que se renuncia al resto de nuestro tiempo de operación. En otros términos, se cede el control al siguiente.

A continuación se muestra la correspondiente salida por pantalla:

```
012010324567891

32000111200301132144422233443352555056677889 9

44563761

46726556677878899


839

8979485

9

6789


Pulse una tecla para continuar...
```

Estamos lejos del resultado esperado.

2. La simulación de la sincronización

La primera solución es muy simple. Consiste en completar nuestra clase. Estas funciones añadidas se traducirán en CLR por un contexto de ejecución particular que se va a encargar de proteger todos los métodos de la clase de ejecuciones actuales.

Añada en encabezado del archivo using System.Runtime.Remoting.Contexts;.

En la definición de la clase Prueba use como prefijo el atributo de "decoración" [Sincronización].

Herede la clase Prueba de la clase ContextBoundObject.

No utilice más las expresiones lambda.

A continuación se muestra el código correspondiente:

```
using System;

using System.Threading;

using System.Runtime.Remoting.Contexts;



namespace PruebaThreadSinSincro

{

  class Program
```

```
  {
    static void Main(string[] args)
    {
      Prueba t = new Prueba();
      t.TratamientoPrincipal();
    }
  }

  [Sincronization]
  class Test: ContextBoundObject
  {
    public void TratamientoPrincipal()
    {
      for (int i = 0; i < 10; i++)
      {
        new Thread(
            new ThreadStart(TratamientoSecundario)).Start();
      }
    }
    public void TratamientoSecundario()
    {
      for (int j = 0; j < 10; j++)
      {
        Console.Write(j);
        Thread.Sleep(0);
      }
      Console.WriteLine();
    }
  }
}
```

A continuación se muestra la salida por la consola correspondiente:

```
0123456789
0123456789
```

```
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
Pulse una tecla para continuar...
```

Este primer método es muy eficaz, pero un poco radical. De hecho, todos los métodos de la clase que hayan recibido estas funciones añadidas no se podrán ejecutar más de manera simultánea, lo cual no resulta muy óptimo.

3. La palabra clave lock

La idea de esta segunda solución es tomar un objeto y utilizarlo como "bloqueo" para limitar el acceso a la sección sensible del código. Esta sección sensible también se llama sección crítica. El objeto en cuestión normalmente es el atributo que queramos proteger; en caso contrario, una instancia de Object es perfectamente válida.

A continuación, la palabra clave lock y sus llaves entran en escena para delimitar la zona a proteger. La sintaxis es muy parecida a la de using, estudiada anteriormente.

```
using System;
using System.Threading;



namespace PruebaThreadSinSincro
{
  class Program
  {
    static void Main(string[] args)
    {
      Prueba t = new Prueba();
      t.TratamientoPrincipal();
    }
  }
```

```
  class Prueba
  {
    private Object bloqueo = new Object();

    public void TratamientoPrincipal()
    {
      for (int i = 0; i < 10; i++)
      {
        new Thread(new ThreadStart(
          () =>
          {
            lock (bloqueo)
            {
              for (int j = 0; j < 10; j++)
              {
                Console.Write(j);
                Thread.Sleep(0);
              }
            }
            Console.WriteLine();
          }
          )).Start();
      }
    }
  }
}
```

La salida por la consola se corresponde con lo esperado.

4. La clase Monitor

Hay que saber que lock (bloqueo){…} es la escritura de acceso directo de un código un poco más complejo, basada en el uso de la clase Monitor.

El siguiente código:

```
lock (bloqueo)
{
  // Tratamiento
}
```

corresponde en C# 4 a:

```
bool bloqueoActivado = false;
try
{
  Monitor.Enter(bloqueo, ref bloqueoActivado);
  // Tratamiento
}
finally
{
  if (bloqueoActivado == true)
    Monitor.Exit(bloqueo);
}
```

Cuando se lee este código, observamos que hay un problema importante que puede aparecer. Imaginemos que durante la operación (marcada en el código // Tratamiento) se produce una excepción y que, para colmo de males, sucede en mitad de una actualización del objeto compartido. En este caso, como sabe, se llamará al código de finally y, como está escrito automáticamente, va a abrir el bloqueo en un objeto compartido que está en un estado totalmente inestable.

Vemos aquí una primera razón para pasar de lock a Monitor. Tan pronto como la operación se vuelve algo complicada tendrá que escribir en el finally el código que permita restaurar el objeto compartido al estado en el que estaba antes de la llamada (acción llamada habitualmente rollback).

Sepa también que Monitor ofrece mucho más control que lock en la sección crítica y los threads que están esperando acceder. Por ejemplo, su método TryEnter evita estar bloqueado hasta que el objeto se libere por otro thread.

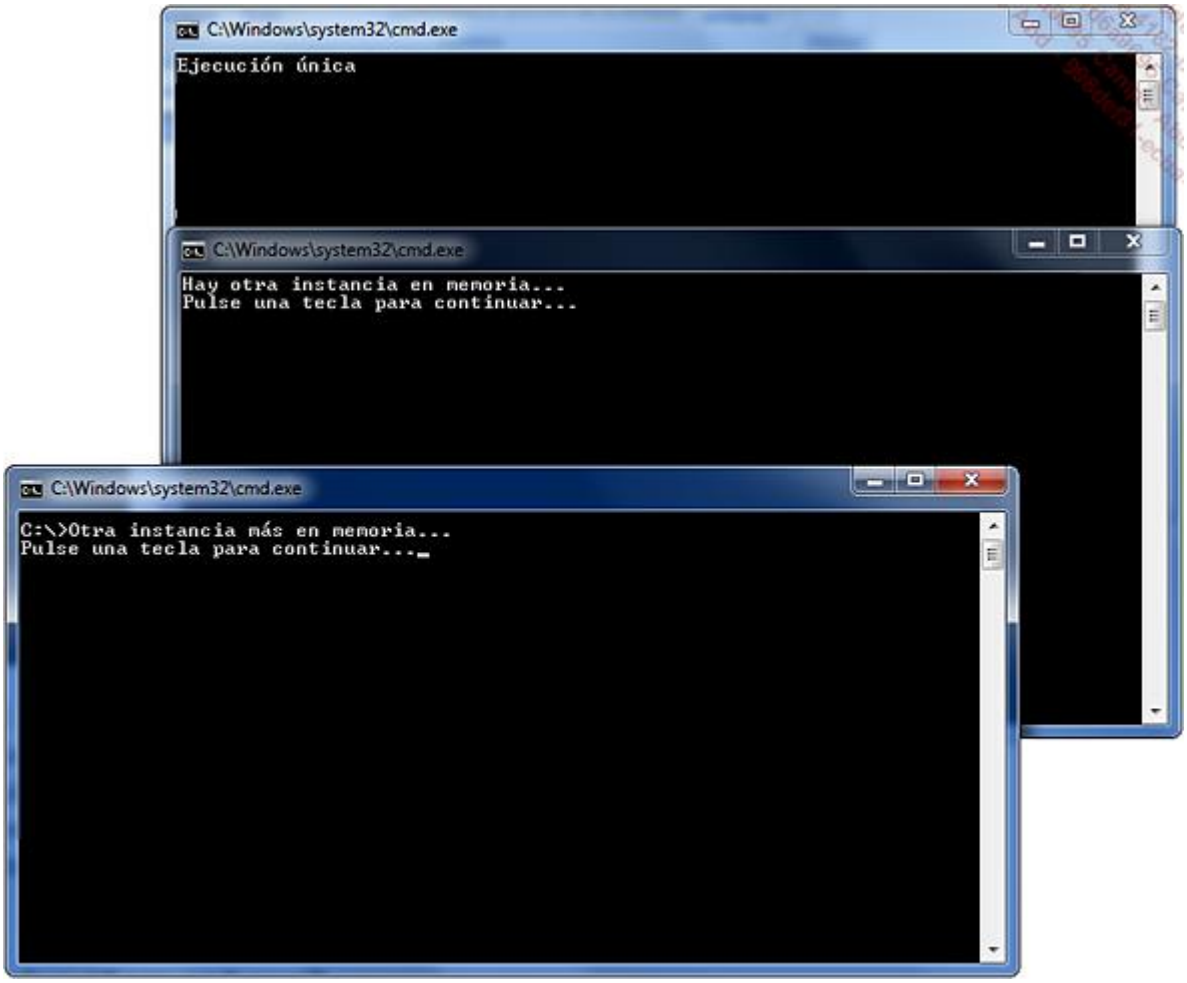
5. La clase Mutex

Un mutex es un "super Monitor", en el sentido de que es capaz de impedir la ejecución simultánea de secciones críticas entre varios procesos. Puede utilizar un mutex para vincular su aplicación a una única instancia.

A continuación se muestra un ejemplo de código que realiza esta función:

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace SynchroInterThreads
{
  class Program
  {
    static void Main(string[] args)
    {
      using (var mutex
        = new Mutex(false,
                    "MiEmpresa.MiServicio.MiPrograma"))
      {
        // Esperamos un poco si alguna otra instancia se está cerrando
        if (!mutex.WaitOne(TimeSpan.FromSeconds(3), false))
        {
          Console.WriteLine(
            "Hay otra instancia en memoria...");
          return;
        }
        RunProgram();
      }
    }
```

```
    static void RunProgram()
    {
      Console.WriteLine("Ejecución única");
      Console.ReadLine();
    }
  }
}
```



El mutex es visible por todo el sistema operativo gracias al nombre que tiene. Una aplicación .NET y una aplicación no gestionada podrían utilizar un mutex con el mismo nombre para, por ejemplo, limitar el acceso a un periférico físico. Un mutex consume muchos recursos y es muy lento en ejecución. Su elección respecto a un Monitor debe estar muy justificada.

6. La clase Semaphore

Podemos considerar el Semaphore como un Mutex que permite un número definido de pases. Un Mutex simplemente es un Semaphore con "un hueco".

Comunicación entre threads

En realidad, Lock y Monitor no establecen una comunicación inter-threads. Protegen la aplicación de la ejecución simultánea de secciones críticas y sus secciones críticas se pueden llamar en cualquier momento.

1. Join

El método Join permite a un thread principal "dormirse" mientras espera el final de la ejecución de un thread secundario.

Ejemplo de código:

```
using System;
using System.Threading;
namespace SynchroInterThreads
{
  class Program
  {
    static void Main(string[] args)
    {
      Prueba t = new Prueba();
      t.TratamientoPrincipal();
    }
  }
  class Prueba
  {
    public void TratamientoPrincipal()
    {
      Console.WriteLine("Inicio TratamientoPrincipal");
      ThreadStart ts
        = new ThreadStart(TratamientoSecundario);
      Thread t = new Thread(ts);
```

```
      t.IsBackground = false;
      t.Priority = ThreadPriority.Highest;
      t.Name = "Es mi thread :)";
      t.Start();

      t.Join();

      Console.WriteLine("Fin TratamientoPrincipal");
    }

    private void TratamientoSecundario()
    {
      Console.WriteLine("Inicio TratamientoSecundario");
      Thread.Sleep(1000 * 10);
      Console.WriteLine("Fin TratamientoSecundario");
    }
  }

}
```

Salida por pantalla asociada:

```
Inicio TratamientoPrincipal
Inicio TratamientoSecundario
Fin TratamientoSecundario
Fin TratamientoPrincipal
Pulse una tecla para continuar...
```

Salida pantalla asociada sin la línea t.Join();:

```
Inicio TratamientoPrincipal
Fin TratamientoPrincipal
Inicio TratamientoSecundario
Fin TratamientoSecundario
Pulse una tecla para continuar...
```

Un thread dormido o bloqueado no consume tiempo de máquina.

El uso del método Join es muy eficaz cuando queremos sincronizarnos al final de una operación completa.

2. Las sincronización de eventos

Si queremos sincronizarnos en determinadas fases de una operación secundaria, el método Join no se puede utilizar. Por ejemplo, imaginemos un thread A encargado de recuperar tramas de bytes y un thread B responsable de procesarlas. El thread A no se va a detener después de la primera trama recibida; va a continuar recibiendo otros bytes y los va a preparar para formar la siguiente trama. Por tanto, el thread B no puede utilizar un Join porque el thread A no tiene fin programado. Para resolver esto, solo tenemos una única forma de proceder, que consiste en utilizar un event/delegate clásico. Aquí, el thread B está "dormido" y se debe despertar cuando el thread A tenga una trama lista para procesarse.

El objeto a utilizar para hablar entre el thread A y el thread B se llama evento de sincronización (sincronization event), y no hay que confundirlo con los objetos de tipo event estudiados anteriormente.

Un evento de sincronización tiene dos estados posibles: notificado y no notificado y el gestor de threads restaura un thread dormido cuando el evento que le interesa pasa a estado notificado.

En nuestro ejemplo:

- Thread A y thread B comparten un mismo evento de sincronización.
- Thread B, cuando no tiene nada más que tratar, se queda dormido en la llamada al método WaitOne del evento de sincronización.
- Después de un tiempo dado, Thread A, que ha recibido una trama completa, la pone en la fila y llama al método Set del objeto de sincronización.
- La CLR despierta al thread B, que accede a la fila de espera de las tramas disponibles, para recuperar la nueva trama recibida. Durante este tiempo, thread A construye la siguiente.

Si el evento de sincronización es de tipo AutoResetEvent, entonces pasa inmediatamente al estado no notificado durante la recuperación del thread B.

Si es de tipo ManualResetEvent, entonces el thread B deberá llamar al método del evento de sincronización Reset para que pase "manualmente" a modo no notificado.

Ejemplo de código con sincronización entre dos threads:

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace SynchroInterThreads
{
  class Program
  {
    static void Main(string[] args)
    {
      Prueba t = new Prueba();
      t.TratamientoPrincipal();
    }
  }

  class Prueba
  {
    private AutoResetEvent eventoTramaRecibida
      = new AutoResetEvent(false);
    private bool termina=false;

    private Queue<int> archivoTramas = new Queue<int>();

    public void TratamientoPrincipal()
    {
      Console.WriteLine("Inicio TratamientoPrincipal");

      Thread threadA = new Thread(
        new ThreadStart(LecturaBytes));
      threadA.Name = "Thread LecturaBytes";
      threadA.Start();
```

```
      Thread threadB = new Thread(
        new ThreadStart(LecturaTramas));
      threadB.Name = "Thread LecturaTramas";
      threadB.Start();

      Console.WriteLine("Pulse en Intro para parar");
      Console.ReadLine();
      // Actualización de un booleano que se comprueba en ambos threads
      // indicando el final del trabajo
      termina = true;

      // Espera que los dos threads terminen
      threadA.Join();
      threadB.Join();

      Console.WriteLine("Fin TratamientoPrincipal");
    }

    private void LecturaBytes()
    {
      while (!termina)
      {
        // Simula una recepción de una trama
        int i = 0;
        for(; i<100 && !termina;i++)
          Thread.Sleep(5);
        Console.WriteLine("Trama recibida");
        // Pone en la fila la trama recibida
        archivoTramas.Enqueue(i);
        // Indica al threadB que puede leer la trama
        eventoTramaRecibida.Set();
      }
    }
```

```
    private void LecturaTramas()
    {
      while (!termina)
      {
        if (eventoTramaRecibida.WaitOne())
        {
          // eventoTramaRecibida es de tipo AutoResetEvent
          // y pasa inmediatamente al estado no notificado
          int numBytes = archivoTramas.Dequeue();
          Console.WriteLine("Trama decodificada " + numBytes);
        }
      }
    }
  }
}
```

Salida correspondiente por la consola:

```
Inicio TratamientoPrincipal
Pulse en Intro para parar
Trama recibida
Trama decodificada 100
Trama recibida
Trama decodificada 100
Trama recibida
Trama decodificada 100
Trama recibida
Trama decodificada 100

Trama recibida
Trama decodificada 24
Fin TratamientoPrincipal
Pulse una tecla para continuar...
```

El tipo AutoResetEvent es muy práctico. Entonces, ¿por qué no utilizarlo siempre?

Sencillamente porque varios threads pueden estar esperando un evento de sincronización. Si el primer thread llamado pone al objeto en estado no notificado, el resto permanecerán dormidos. Esta es la razón por la que existe el tipo ManualResetEvent, que permite restablecer "manualmente" al estado no notificado.

Varios threads pueden esperar un mismo evento y un thread puede esperar varios eventos.

Es posible despertar al thread:

- tan pronto como un evento de la lista esté notificado,
- cuando todos los eventos de la lista están notificados.

Los eventos se deben crear y referenciar en una tabla de tipo WaitHandle[]. El tipo WaitHandle ofrece un método de tipo static, llamado WaitAny, para esperar un evento y WaitAll para esperar que todos los eventos estén notificados.

Usemos esta solución para mejorar nuestro programa anterior. Puede que haya observado que existe un pequeño problema en el método LecturaTramas(), del thread B. El bucle comprueba el valor del booleano termina para saber si se está al final del uso de la aplicación. Si no es el caso, "se duerme" esperando que se firme un thread A. El thread A nunca se duerme; simula una recepción continua y, en cada byte, comprueba el booleano termina. Si la variable pasa a true, sale del bucle de operación y notifica el evento "trama recibida" (mientras que la trama no se ha recibido totalmente), para que el thread B no se quede eternamente dormido. La salida por la consola muestra el tamaño de la última trama decodificada, que es de un tamaño diferente al normal.

Para resolver el problema, vamos a:

- Sustituir el booleano del thread principal por un event llamado eventoAbandono. Como este event se comprobará por los threads A y B, tendrá que ser de tipo ManualResetEvent.
- Modificar el thread A para que compruebe sin dormirse la notificación de eventoAbandono. Esto es posible gracias a la sobrecarga del método WaitOne, que recibe como argumento un tiempo de espera.
- Modificar el thread A para que solo produzca el evento "trama recibida " únicamente cuando la trama se haya recibido completamente.
- Hacer que el thread B esté receptivo al mismo tiempo al evento "trama recibida" y al evento "abandono" gracias al método WaitAny de WaitHandle[].

Código fuente del programa modificado:

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace SynchroInterThreads
{
  class Program
  {
    static void Main(string[] args)
    {
      Prueba t = new Prueba();
      t.TratamientoPrincipal();
    }
  }

  class Prueba
  {
    // El evento eventoTramaRecibida solo se monitoriza
    // por un thread, por lo que puede ser automático
    private AutoResetEvent eventoTramaRecibida
      = new AutoResetEvent(false);

    // El evento abandono se va a monitorizar
    // por los dos threads, por lo que debe ser
    // de tipo ManualResetEvent
    private ManualResetEvent eventoAbandono
          = new ManualResetEvent(false);

    private Queue<int> archivoTramas = new Queue<int>();

    public void TratamientoPrincipal()
    {
```

```
    Console.WriteLine("Inicio TratamientoPrincipal");

    Thread threadA = new Thread(
      new ThreadStart(LecturaBytes));
    threadA.Name = "Thread LecturaBytes";
    threadA.Start();

    Thread threadB = new Thread(
      new ThreadStart(LecturaTramas));
    threadB.Name = "Thread LecturaTramas";
    threadB.Start();

    Console.WriteLine("Pulse en Intro para parar");
    Console.ReadLine();

    // Notifica el evento de abandono
    eventoAbandono.Set();

    // espera a que los dos threads se cierren automáticamente
    threadA.Join();
    threadB.Join();

    Console.WriteLine("Fin TratamientoPrincipal");
  }

  private void LecturaBytes()
  {
    while (true)
    {
      int i = 0;
      for (; i < 100; i++)
      {
        Thread.Sleep(5);
```

```
        // Comprueba el estado del evento de abandono
        // SIN estar bloqueado (duración de la espera = 0)
        if (eventoAbandono.WaitOne(0))
          return;
      }
      Console.WriteLine("Trama recibida");
      // Pone en la fila la trama
      archivoTramas.Enqueue(i);
      // Indica al threadB que puede leer la trama
      eventoTramaRecibida.Set();
    }
  }

  private void LecturaTramas()
  {
    // Creación de una tabla de WaitHandle que contiene
    // las referencias a los dos eventos
    WaitHandle[] tableEvents = new WaitHandle[]
    {
      eventoTramaRecibida,
      eventoAbandono
    };

    while (true)
    {
      switch (WaitHandle.WaitAny(tableEvents))
      { // La salida de WaitAny contiene el índice
        // del evento en la tabla que fue notificado
        case 0: // corresponde al eventoTramaRecibida
          // El evento es de tipo automático, por lo que
          // no hay que hacer nada para reiniciarlo
          int numBytes = archivoTramas.Dequeue();
          Console.WriteLine("Trama decodificada " + numBytes);
```

```
            break;
          case 1: // corresponde al eventoAbandono
            return;
        }
      }
    }
  }
}
```

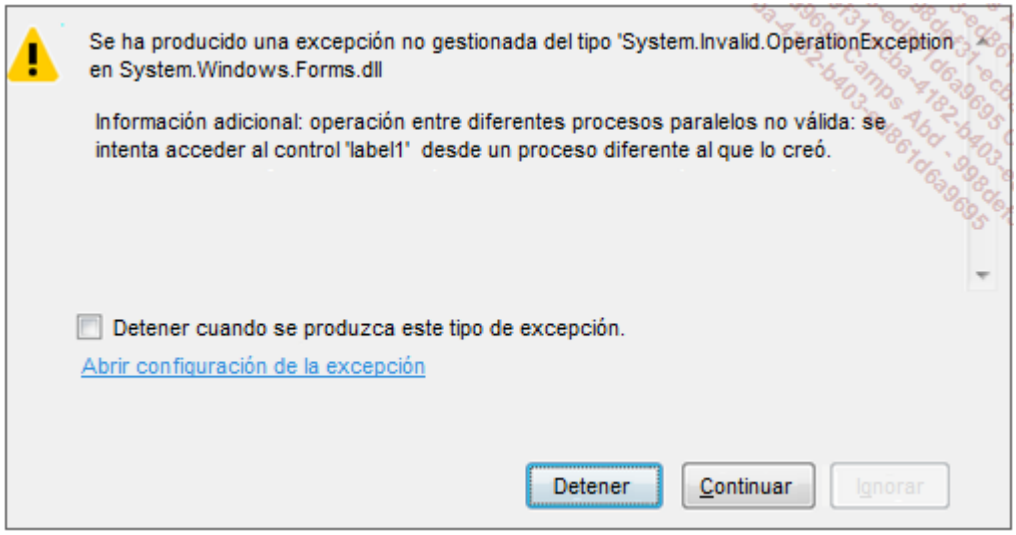
Salida por la consola:

```
Inicio TratamientoPrincipal
Pulse en Intro para parar
Trama recibida
Trama decodificada 100
Trama recibida
Trama decodificada 100
Trama recibida
Trama decodificada 100
Trama recibida
Trama decodificada 100

Fin TratamientoPrincipal
Pulse una tecla para continuar...
```

3. Comunicación entre threads secundarios e IHM

Windows Forms y Windows Presentation Foundation permiten crear interfaces gráficas ricas. Estas interfaces gráficas (IHM, que significa "interfaces hombre-máquina" o incluso UI, que significa User Interface) están fuertemente acopladas a los threads que las crean. Este thread se encarga de gestionar la visualización y actualizar los componentes gráficos. Este thread no es una excepción a la regla y se va a gestionar como el resto, pudiendo dejar un objeto en plena actualización. Por lo tanto, se corre el riesgo de intentar modificar directamente los componentes gráficos desde un thread diferente al que sirve para crear el formulario.

A continuación se muestran el tipo de situación que se puede producir:



Para resolver este problema de comunicación entre threads, los desarrolladores Win32 tuvieron la idea de utilizar mensajes Windows, que se enviarán por el thread secundario y que serán obviados por la ventana del formulario. El defecto de esta solución es que los medios de transmisión (WPARAM y LPARAM) son débilmente tipados.

Afortunadamente, C# ofrece una solución eficaz al problema con dos métodos Invoke y BeginInvoke, que se implementan en cada control Windows Forms y en el Dispatcher de WPF. Cuando un thread secundario debe modificar un control gráfico del IHM, llama al método Invoke o BeginInvoke del control, pasándole la operación a realizar en forma de delegate, eventualmente seguido de una tabla de argumentos. De esta manera, el trabajo de actualización se realiza por el thread responsable del IHM. Si se utilizada el método Invoke, el thread secundario espera a que termine la actualización del control, si este último es asíncrono.

Para ilustrar esto, a continuación se muestra un formulario escrito con Windows Forms que muestra una información resultante de un thread cada segundo.

En esta primera versión, el formulario ofrece un método de actualización del control que gestiona la invocación en caso de que el thread que lo utiliza no sea el que ha creado el control. Para ello, se comprueba la propiedad InvokeRequired del control y, si la invocación es necesaria, se llama a este mismo método pero esta vez desde el thread creador. Todo esto es posible gracias al uso del delegate creado para la ocasión y llamado UpdateLabel.

```
using System;

using System.Threading;

using System.Windows.Forms;


namespace SynchroIHM

{

  public partial class Form1: Form

  {
```

```
    private delegate void UpdateLabel(String nuevoValor);
    public Form1()
    {
      InitializeComponent();

      Thread t = new Thread( new ThreadStart(
        () =>
          {
            int contador = 0;
            while (true)
            {
              Thread.Sleep(1000);
              UpdateLabel1(contador.ToString());
              contador++;
            }
          }
          ));
      t.IsBackground = true;
      t.Start();
    }

    private void UpdateLabel1(String nuevoValor)
    {
      if( label1.InvokeRequired )
        label1.Invoke(
          new UpdateLabel(UpdateLabel1),
          new object[] { nuevoValor }
          );
      else
        label1.Text = nuevoValor;
    }
  }
}
```

Si sabe que el método UpdateLabel1 solo se llamará desde el thread t, el código se podría simplificar como sigue.

UpdateLabel1 desaparece y es el thread el que realiza la invocación, creando un método anónimo.

```
using System;
using System.Threading;
using System.Windows.Forms;
namespace SynchroIHM
{
  public partial class Form1: Form
  {
    public Form1()
    {
      InitializeComponent();
      Thread t = new Thread(new ThreadStart(
        () =>
        {
          int contador = 0;
          while (true)
          {
            Thread.Sleep(1000);
            label1.Invoke((MethodInvoker)delegate
            {
              label1.Text = contador.ToString();
            });
            contador++;
          }
        }
          ));
      t.IsBackground = true;
      t.Start();
    }
  }
}
```

4. Ejercicio

a. Enunciado

Para establecer en la aplicación la comunicación entre el thread secundario y el thread IHM, va a:

Escribir un formulario que contenga una barra de progreso, que vaya de cero a cincuenta, y dos botones dispuestos como se muestra a continuación.

Asociar al clic del botón **Start** un gestor de eventos que permita lanzar un thread. Este thread llamará cada 500 milisegundos a un método del formulario que permite incrementar la posición de la barra de progreso. Este lapso de tiempo será un argumento que se pase durante la ejecución del thread.

Escribir el método que permite incrementar la posición de tal manera que se pueda llamar desde el thread principal o desde el thread secundario.

Asociar al clic del botón **Stop** un gestor de eventos, que permita parar el thread actual.

b. Corrección

```
using System;
using System.Threading;
using System.Windows.Forms;
namespace LabSynchroIHM
{

  public partial class Form1: Form
  {
    // Referencia al objeto thread
    // para permitir controlarlo
    private Thread th;

    private bool isAbandon = false;

    public Form1()
    {
```

```
      InitializeComponent();

      progressBar1.Minimum = 0;
      progressBar1.Maximum = 50;
    }

    // Gestor de eventos del botón start
    private void OnButtonStart(object sender, EventArgs e)
    {
      // Inicializa el flag de salida, que se comprobará
      // en el thread  en cada vuelta
      isAbandon = false;
      // Instanciación del thread en modo paso de argumentos
      th = new Thread(
              new ParameterizedThreadStart(BucleAnimacion));
      // Inicia el thread con la duración como argumento
      th.Start(500);
    }

    // Método llamado por el thread
    private void BucleAnimacion(object o)
    {
      // Recuperación del argumento duración
      int duracionPausa = (int)o;
      // Bucle infinito
      while (!isAbandon)
      {
        // Llamada al método de actualización
        AdvanceProgressBar();
        // Pausa
        Thread.Sleep(duracionPausa);
      }
    }
```

```
    // delegate que se encargará de la actualización de la barra
    // por el thread principal
    delegate void advanceProgressBar();
    // Método de actualización de la barra que se puede llamar
    // desde cualquier thread (threadsafe)
    private void AdvanceProgressBar()
    {
      // ¿Es necesaria la invocación?
      if (progressBar1.InvokeRequired)
      {
        // Sí : entonces el trabajo se realizará más tarde
        progressBar1.Invoke(
          new advanceProgressBar(AdvanceProgressBar));
      }
      else
      {
        // No : entonces la barra de progreso avanza
        progressBar1.Value++;
        // y cuando llegue al final, vuelve a empezar en 0
        if (progressBar1.Value >= progressBar1.Maximum)
          progressBar1.Value = 0;
      }
    }

    // Gestor de eventos del botón stop
    private void OnButtonStop(object sender, EventArgs e)
    {
      // Actualización del flag de salida, que se comprueba
      // en el bucle del thread
      isAbandon = true;
    }
  }
}
```

Parte del código generado por Visual Studio:

```
namespace LabSynchroIHM
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components
                                    = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources
        /// should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
```

```
        this.progressBar1 = new System.Windows.Forms.ProgressBar();
        this.button1 = new System.Windows.Forms.Button();
        this.button2 = new System.Windows.Forms.Button();
        this.SuspendLayout();
        //
        // progressBar1
        //
        this.progressBar1.Location
                     = new System.Drawing.Point(24, 49);
        this.progressBar1.Name = "progressBar1";
        this.progressBar1.Size = new System.Drawing.Size(237, 29);
        this.progressBar1.TabIndex = 0;
        //
        // button1
        //
        this.button1.Location = new System.Drawing.Point(24, 102);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(115, 23);
        this.button1.TabIndex = 1;
        this.button1.Text = "Start";
        this.button1.UseVisualStyleBackColor = true;
        this.button1.Click
             += new System.EventHandler(this.OnButtonStart);
        //
        // button2
        //
        this.button2.Location = new System.Drawing.Point(145, 102);
        this.button2.Name = "button2";
        this.button2.Size = new System.Drawing.Size(116, 23);
        this.button2.TabIndex = 1;
        this.button2.Text = "Stop";
        this.button2.UseVisualStyleBackColor = true;
        this.button2.Click
```

```
            += new System.EventHandler(this.OnButtonStop);
      //
      // Form1
      //
      this.AutoScaleDimensions
            = new System.Drawing.SizeF(6F, 13F);
      this.AutoScaleMode =
            System.Windows.Forms.AutoScaleMode.Font;
      this.ClientSize = new System.Drawing.Size(284, 136);
      this.Controls.Add(this.button2);
      this.Controls.Add(this.button1);
      this.Controls.Add(this.progressBar1);
      this.Name = "Form1";
      this.Text = "LabSynchroIHM";
      this.ResumeLayout(false);

        }

        #endregion

        private System.Windows.Forms.ProgressBar progressBar1;
        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.Button button2;
    }
}
```

La programación asíncrona

El problema de la notificación mediante función de llamada (callback) es la función de llamada en sí. De hecho, se ejecuta desde un thread diferente y no se puede hacer lo que se quiera.

Desde la versión 4.5 de .NET (Visual Studio 2012) aparecieron las palabras clave async y await, que han simplificado considerablemente la programación asíncrona. De hecho, este dúo permite transformar a bajo coste los métodos bloqueantes y "lentos" y dotarlos de un funcionamiento asíncrono (por tanto no bloqueante) para que la aplicación permanezca activa.

1. La palabra clave async

La palabra clave async informa al compilador de que el método siguiente tendrá un modo de ejecución asíncrono. Por convención no obligatoria, el nombre del método termina con Async para que su lectura recuerde a sus usuarios este modo de funcionamiento particular.

Ejemplo:

```
private async void TrabajoLargoAsinc(){...
```

2. Contenido de un método async

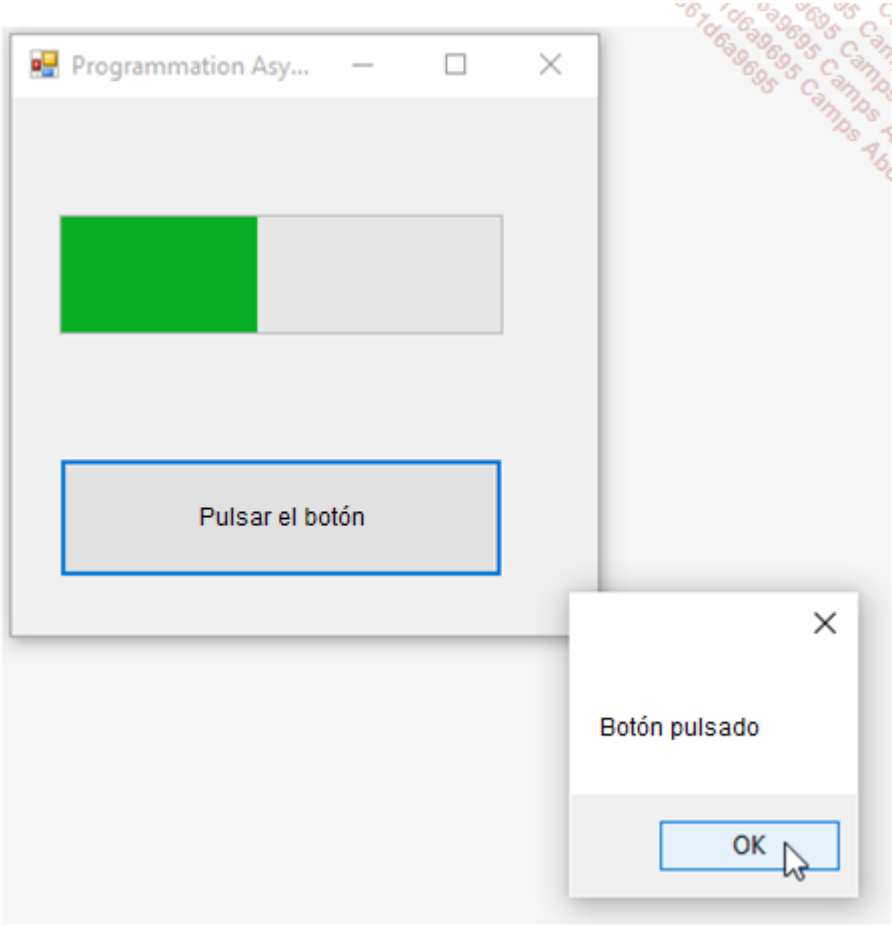
Un método async tiene un comportamiento "síncrono" hasta que su ejecución encuentra una línea que empieza con la palabra clave await. En este momento, devuelve el control al código que lo invoca sin que haya terminado y la instrucción seguida de la palabra clave await empieza una ejecución asíncrona.

El código final recuerda al de un sencillo método síncrono. Si el método async se llama desde un gestor de eventos gráfico, incluso será posible intervenir sobre los controles de pantalla, porque el thread de ejecución async es el mismo que su llamador.

3. Evidencias

El siguiente código es el de un formulario que contiene una barra de progreso y un botón. La barra de progreso se va a actualizar en un método asíncrono. Para provocar que la aplicación permanezca reactiva durante este trabajo, puede pulsar el botón y comprobar la visualización del cuadro de diálogo con la barra de progreso, que continuará el avance en segundo plano.

```
using System;
using System.Threading.Tasks;
using System.Windows.Forms;
namespace ProgramacionAsincrona
{
    public partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
            TrabajoLargoAsinc();
        }
        private async void TrabajoLargoAsinc()
        {
            for (int i = 0; i <5; i++)
            {
                for (int j = 0; j <= 10; j++)
                {
                    await Task.Delay(500);
                    progressBar1.Value = j * 10;
                }
            }
        }
        private void button1_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Botón pulsado");
        }
    }
}
```


Programmation Asy...
Pulsar el botón
Botón pulsado
OK

4. Resultados posibles de un método async

Los tipos de resultado soportados por un método async son:

- void
- Task
- Task<TResult>

Un valor de retorno de tipo void se utiliza cuando el código que lo invoca no necesita saber cuándo terminará el trabajo asíncrono que ha iniciado. En el siguiente código, el gestor del evento "clic" del botón llama al método async, que inicia la operación asíncrona y retoma el control inmediatamente.

```
// El método async llamado que devuelve un void no permite
// definir button0_Click de tipo async
private void button0_Click(object sender, EventArgs e)
{
    TrabajoLargoAsincDevolviendoVoid();
    MessageBox.Show("TrabajoLargoAsincDevolviendoVoid iniciado.");
}

private async void TrabajoLargoAsincDevolviendoVoid()
{
    await Task.Delay(3000);
    MessageBox.Show("Sin retorno", "Fin de TrabajoLargoAsincDevolviendoVoid");
}
```

Un retorno de tipo Task se utiliza cuando el código que invoca al método necesita saber cuándo terminará el trabajo asíncrono que ha iniciado, y este trabajo no devuelve información. Cuando un método async devuelve un tipo Task, es posible llamarlo con la palabra clave await, que permite un modo asíncrono en cascada. Es el caso del siguiente código en el que el gestor del evento "clic" del botón se completa con un async que va a permitir tomar el control de la aplicación una vez que se pulsa el botón, esperando al final de TrabajoLargoAsincDevolviendoTask.

```
// El método async llamado que devuelve un objeto Task permite
// definir button1_Click de tipo async y utilizar
// await antes de llamar a TrabajoLargoAsinc
private async void button1_Click(object sender, EventArgs e)
{
    await TrabajoLargoAsincDevolviendoTask();
    MessageBox.Show("Sin retorno",
"Fin de TrabajoLargoAsincDevolviendoTask");
}

private async Task TrabajoLargoAsincDevolviendoTask()
{
    await Task.Delay(3000);
}
```

Un retorno de tipo Task<TResult> se utiliza cuando el código llamador necesita saber cuándo termina el trabajo asíncrono y además necesita recuperar el resultado de este trabajo. Tiene el mismo comportamiento que el anterior, con un código adicional de retorno del método de trabajo.

```
// El método async llamado que devuelve un objeto Task<string>,
// permite definir button2_Click de tipo async y utilizar await
// antes de llamar a TrabajoLargoAsincDevolviendoTaskYString
// y después recuperar su cadena de retorno
private async void button2_Click(object sender, EventArgs e)
{
    var s = await TrabajoLargoAsincDevolviendoTaskYString();
    MessageBox.Show(s,
"Fin de TrabajoLargoAsincDevolviendoTaskYString");
}

private async Task<string>
TrabajoLargoAsincDevolviendoTaskYString()
{
    await Task.Delay(3000);
    return "Viva la Programación Asíncrona";
}
```

Desde C#6, se puede utilizar la palabra clave await en las secuencias try ... catch.

Introducción

Este capítulo se dirige a los desarrolladores C/C++.

Pasar a la programación orientada a objetos no es sencillo para los desarrolladores, y tampoco para los directores de las empresas, que imaginan sus inversiones en software en código nativo reducidas a la nada.

El framework .NET permite evitar eliminar completamente el pasado, permitiendo intercambios entre su mundo, conocido como "gestionado" y el exterior: el mundo no gestionado en código nativo. En consecuencia, pasar a la programación orientada a objetos se podrá hacer de manera progresiva y, por tanto, será menos traumática para todos. Las DLL realizadas en C/C++ con su correspondiente esfuerzo siempre se podrán utilizar en un esquema de programación orientada a objetos.

Sin embargo, preste atención al hecho de que durante la ejecución del código nativo desde el framework .NET se podrá corto-circuitar la seguridad. Las funciones no gestionadas podrán acceder directamente a los recursos del sistema operativo.

1. Recordatorio sobre las DLL no gestionadas

Una DLL no gestionada es un archivo que contiene código repartido en funciones. En la mayoría de casos, estas funciones son accesibles desde el exterior. Por tanto, hablamos de funciones "exportadas". A diferencia de lo que sucede con un archivo .EXE, es imposible ejecutar directamente el código de un archivo .DLL. Para poder aprovechar sus servicios, es necesario que un ejecutable cargue la DLL, busque dinámicamente la dirección de sus funciones "exportadas" y después las invoque. Las direcciones de sus funciones tienen pocos puntos comunes con los delegate de C#. Se trata solo de los punteros al código, sin otras indicaciones sobre el número y tipos de argumentos esperados e incluso, sobre el tipo de retorno.

El interés de las DLL es contener funciones que se puedan compartir de manera sencilla entre varias aplicaciones. Por ejemplo, el código que permite mostrar los cuadros de diálogo de sistema está contenido en User32.DLL y la mayor parte de las aplicaciones utilizan esta misma DLL para mostrarlos.

Recordemos que las DLL también pueden ser de tipo gestionado. En este caso hablamos de ensamblados, que siempre tienen el objetivo de agrupar objetos que se pueden usar de manera sencilla. Las relaciones entre ensamblados y programas .NET son inmediatas y, por tanto, no son estos tipos de DLL los que nos interesan en este capítulo.

En la mayoría de los casos, un programa gestionado solo podrá llamar a funciones nativas (no gestionadas) si están exportadas en DLLs. Si durante una portabilidad determinadas operaciones no gestionadas importantes están codificadas en los ejecutables, entonces habría que exportarlas en las DLL.

2. P-Invoke y su Marshal

La parte del framework .NET encargada de las comunicaciones entre diferentes sistemas se llama Platform Invocation Services o de manera más sencilla P-Invoke. Los desarrolladores Java supieron ver la equivalencia con JNI (Java Native Interface). En la Platform Invocation Services, la parte encargada de las conversiones de formato de variables se llama Marshal.

La programación de una llamada a una función nativa es relativamente sencilla si el formato de los datos intercambiados es "clásico". Hay que conocer las correspondencias entre los objetos CTS (Common Type System) y los tipos de C/C++.

Esto se complica un poco cuando hay que transmitir información más compleja como estructuras, tablas u otros tipos de agregadores. Por tanto, resulta importante entender bien los mecanismos de conversión que se proponen y que vamos a descubrir ahora.

El resto de este capítulo tiene que ver con la comunicación entre C# y C, porque representa el caso más clásico. Hay otros métodos para interfasar con clases C++ escritas en Visual Studio (C++ Interop) e incluso con objetos COM.

La presentación de la programación del servicio P-Invoke se basa aquí en una serie de ejemplos de complejidad creciente. El código de estos ejemplos se puede descargar desde la página Información. Al inicio de las explicaciones de cada ejemplo se dará el nombre de su proyecto correspondiente.

En las explicaciones que siguen se utilizará habitualmente el término de código C/C++. De hecho, se trata de código C que aprovecha las mejoras añadidas por el compilador C++, como el tipo bool o la declaración simplificada de las estructuras.

El caso sencillo

El primer ejemplo (proyecto PInvoke01 para descargar), muestra el caso más sencillo. El objeto C# llama a una función C/C++ que no espera ningún argumento y que no devuelve nada; por tanto, no habrá problemas de conversión de tipos entre los dos mundos. Sin embargo, este caso "sencillo" va a mostrar cómo cargar una DLL no gestionada, cómo encontrar la entrada de la función expuesta y después cómo llamarla.

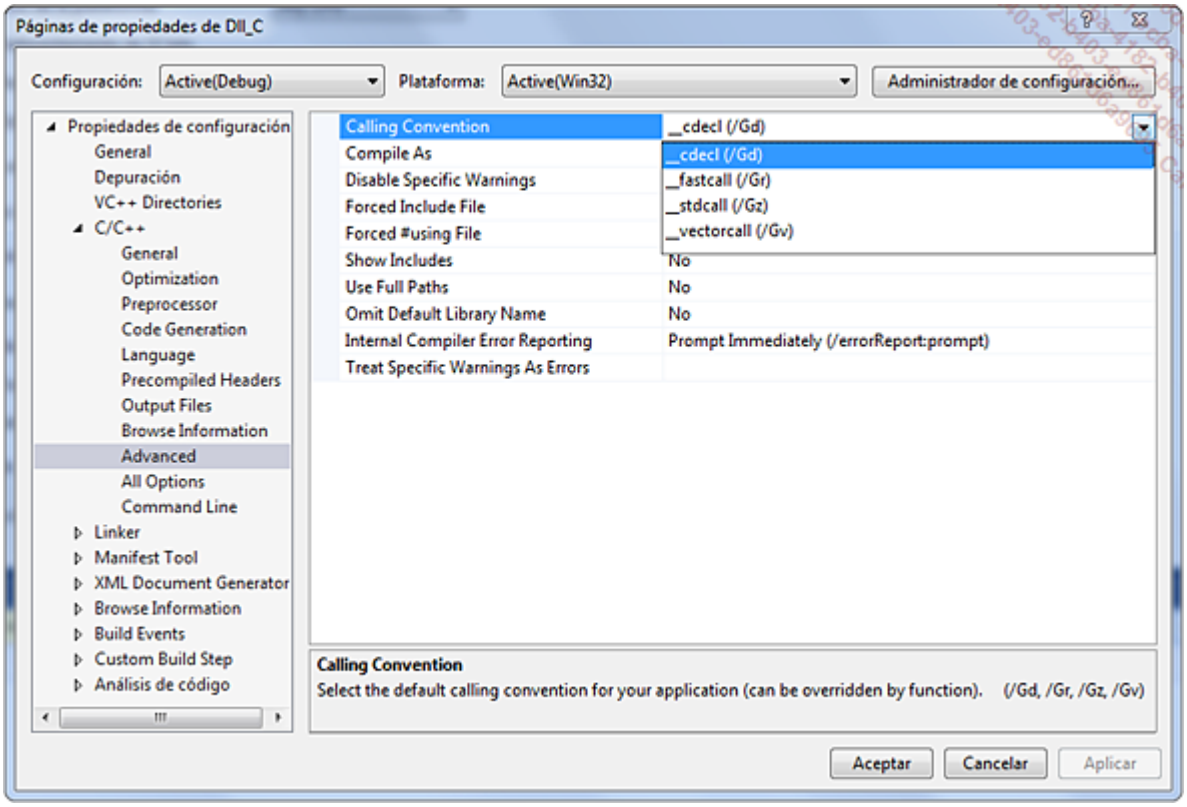
1. Declaración y llamada

Por parte de la DLL, la función C llamada FuncionNoGestionada se debe declarar como "expuesta". En el entorno de desarrollo de Microsoft y para la mayor parte de los compiladores GCC debe estar precedida por __declspec(dllexport) y declararse en el archivo .DEF del proyecto.

```
#include "stdafx.h"
#include "Dll_C.h"



__declspec(dllexport) void FuncionNoGestionada(void)
{
    MessageBeep(MB_ICONASTERISK);
}
LIBRARY
EXPORTS

FuncionNoGestionada
```

Durante una llamada a una función, se utiliza la pila para preguntar los argumentos y guardar la dirección de retorno. La manera en que se gestiona la pila se llama **convención de llamada**. Hay varias técnicas y una mala adecuación entre el código que invoca y el código invocado provoca funcionamientos incorrectos, normalmente difíciles de entender.

Por defecto, Visual Studio utiliza la convención __cdecl, que consiste en disponer los argumentos de derecha a izquierda y solicitar al código que invoca que reinicie la pila hasta el retorno de la función. En caso necesario, este ajuste por defecto se puede modificar en la página de las propiedades del proyecto de la DLL.



Las funciones "Windows" constituyen la API Win32 y se utiliza la convención __stdcall, que consiste en situar en la pila los argumentos de derecha a izquierda y a petición de la llamada, para reinicializar la pila antes de devolver al código que invoca.

Visual Studio permite definir la convención de llamada por defecto en la página de las propiedades del proyecto DLL, pero es posible redefinir esta convención por función.

El caso "sencillo" también es cuando se dispone de todos los archivos fuente de la DLL. Normalmente estamos obligados a interfasar con las DLL relacionadas con los equipamientos y, por tanto, que solo dispongan de la versión binaria. El tipo de convención de llamada se debería encontrar en la documentación del constructor. Si no es el caso, hay que armarse de paciencia y hacer varios intentos.

Dedicado a los desarrolladores de Windows CE: la version .NET "Compact Framework" utilizada en los terminales CE solo soporta la convención __cdecl.

Por parte de C#, ahora se debe definir dónde y cómo acceder a la función C/C++. Para ello, hay que:

- Declarar el prototipo de la función, precedido por las palabras clave static y extern.
- Asociar al prototipo una función añadida que va a declarar dónde y cómo utilizar la función. En nuestro ejemplo la función se publica en la DLL llamada "DLL_C.DLL", su nombre es FuncionNoGestionada y su convención de llamada es __cdecl.

```
DllImport("Dll_C.dll", EntryPoint = "FuncionNoGestionada",
CallingConvention = CallingConvention.Cdecl)]
static extern void FuncionNoGestionada();
```

El nombre de la función utilizada en el atributo EntryPoint de DllImport debe corresponder con el nombre de la función exportada en la DLL. Sin embargo, la variable asociada puede tener otro nombre, aunque generalmente se utiliza el mismo.

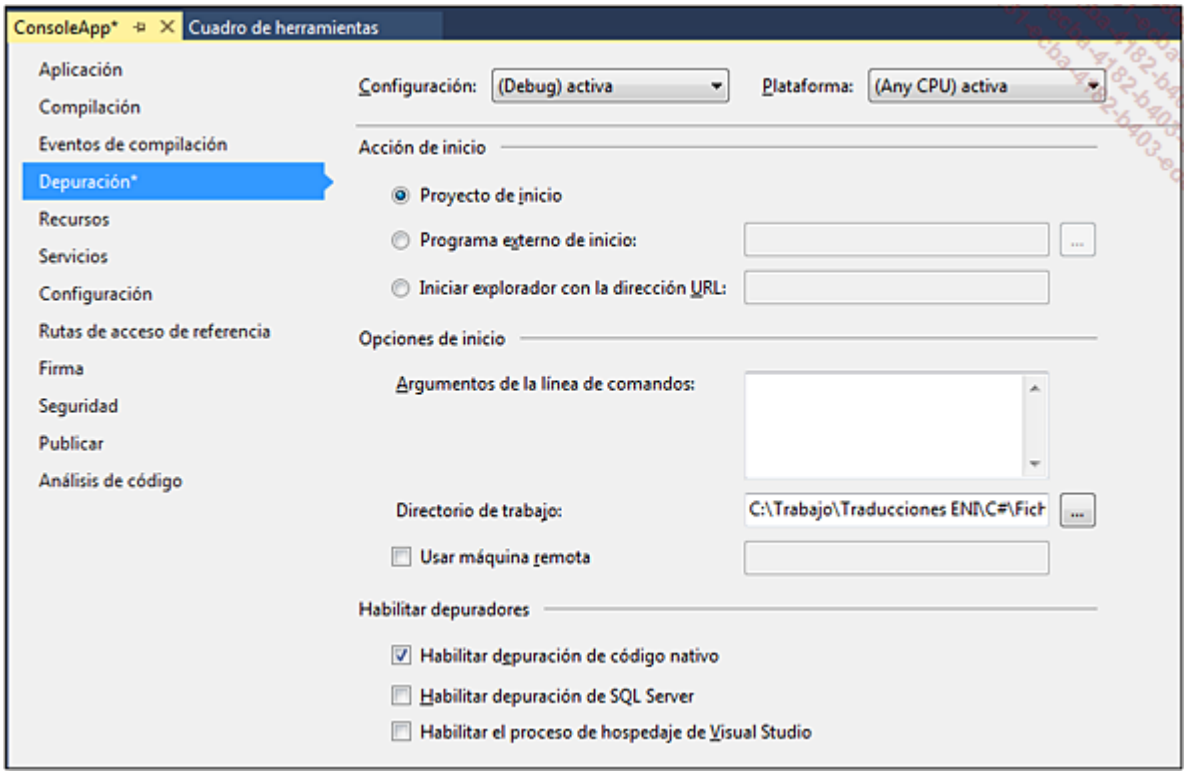
A continuación la función se puede llamar desde una clase C#. Esto genera el código siguiente:

```
using System.Runtime.InteropServices;


namespace ConsoleApp
{
    class Program
    {
        [DllImport("Dll_C.dll", EntryPoint = "FuncionNoGestionada",
CallingConvention = CallingConvention.Cdecl)]
        static extern void FuncionNoGestionada();


        static void Main(string[] args)
        {
            FuncionNoGestionada();
        }
    }
}
```

Compile y lance la aplicación de consola C#. Esto debe hacer "beep".

2. Ajuste de Visual Studio para la puesta a punto

El entorno de desarrollo permite la puesta a punto de aplicaciones C# que utilizan funciones C. Por defecto la opción **Habilitar depuración de código nativo** no está activa en el proyecto C#, lo que hace imposible trazar paso a paso las instrucciones de C llamadas por C#. Para validar este modo de puesta a punto vaya a las propiedades del proyecto C#, página **Depuración**, y marque la opción **Habilitar depuración de código nativo**.



Llamada con argumentos y retorno de función

Este segundo ejemplo (proyecto PInvoke02 para descargar) muestra:

- una transmisión de argumentos de tipos clásicos entre un objeto C# y una función C/C++,
- la recuperación del resultado de la función C/C++ en el objeto C#.

La función C/C++ devuelve la suma de dos enteros que se pasan como argumentos.

P-Invoke tiene automáticamente en cuenta el tipo int, y también lo hace su brazo derecho, el Marshal; por lo tanto, por el momento, no hay nada demasiado complicado.

Por parte de la DLL, a continuación se muestra la función C/C++:

```
#include "stdafx.h"
#include "Dll_C.h"



__declspec(dllexport) int Sumar(int x, int y )
{
    return x + y ;
}
```

Piense siempre en declarar la función en el archivo de definición (.DEF):

```
LIBRARY
EXPORTS

Sumar
```

Por parte de C#, a continuación se muestra la declaración y la llamada a la función C/C++:

```
using System;
using System.Runtime.InteropServices;

namespace ConsoleApp
{
    class Program
    {
        [DllImport("Dll_C.dll", EntryPoint = "Sumar",
CallingConvention = CallingConvention.Cdecl)]
        static extern int Sumar(int x, int y );

        static void Main(string[] args)
        {
            Console.WriteLine( Sumar(4, 5));
        }
    }
}
```

Después de la compilación, compruebe el funcionamiento con [Ctrl][F5]:

"9" se muestra correctamente, por lo que el resultado del cálculo no gestionado se ha transmitido correctamente al código gestionado.

No ha sido necesaria ninguna línea de código de conversión. Los argumentos utilizados se transfieren directamente de un mundo al otro. Sin embargo, el problema es que los nombres de los tipos C# no se corresponden necesariamente sus homólogos en C/C++, sobre todo

teniendo en cuenta que la API Windows rebautiza a la mayor parte de estos tipos haciendo uso de muchos **typedef** en **WinNT.h**.

A continuación se muestra una tabla resumen de las equivalencias:

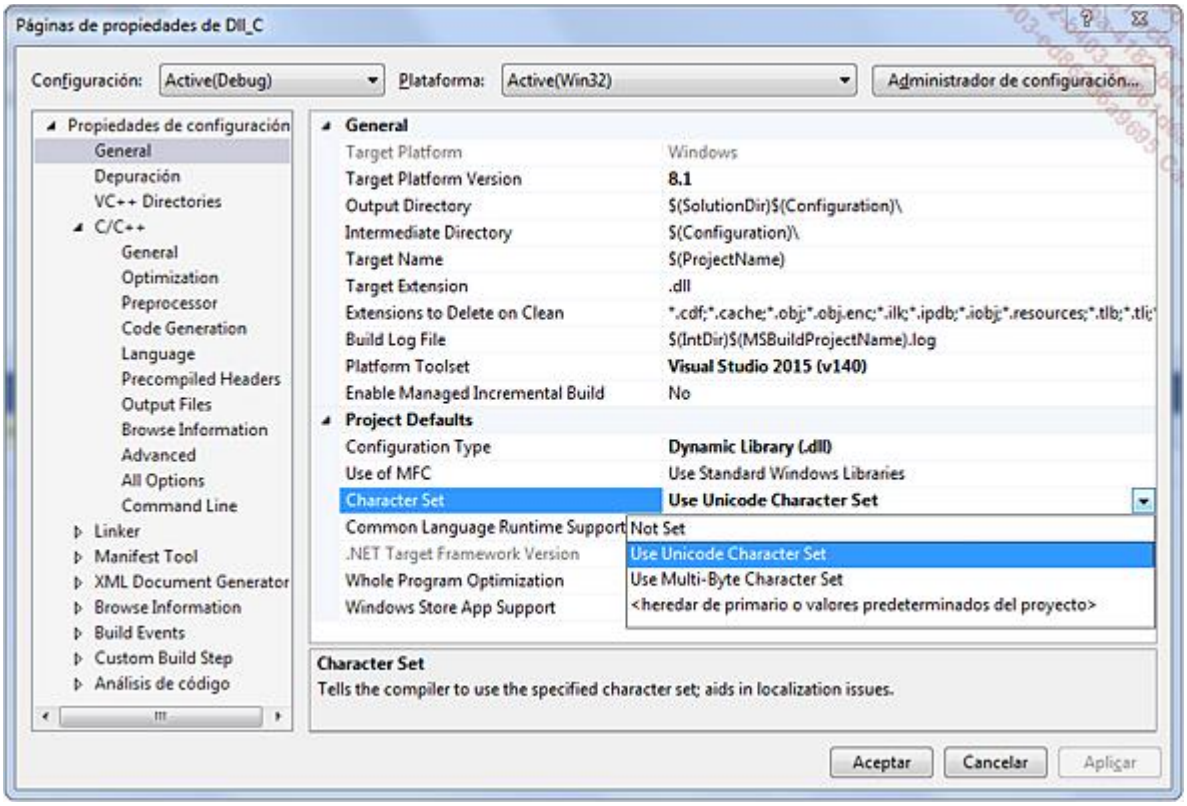
Tipo C/C++	Formato	Tipo Windows	Tipo C#	Tipo gestionado
char	byte con signo	CHAR	sbyte	System.SByte
unsigned char	byte sin signo	BYTE	byte	System.Byte
short	entero con signo 16 bits	SHORT	short	System.Int16
unsigned short	entero sin signo 16 bits	WORD	ushort	System.UInt16
int / long	entero con signo 32 bits	INT / LONG	int	System.Int32
unsigned int	entero sin signo 32 bits	DWORD	uint	System.UInt32
long long	entero con signo 64 bits	INT64	long	System.Int64
unsigned long long	entero sin signo 64 bits	UINT64	ulong	System.UInt64
float	entero de coma flotante	FLOAT	double	System.Double

Como el tipo de los argumentos intercambiados solo afecta a esta tabla, no hay nada particular que añadir.

Tratamiento con las cadenas de caracteres

1. Codificación de los caracteres

Los problemas empiezan con la operación con caracteres. Como se ha recordado con anterioridad, la codificación de los caracteres puede utilizar formatos muy diferentes. El tipo char del C/C++ codifica el carácter con un byte (8 bits, es decir 256 combinaciones). Esta codificación minimalista que deja, entre otros, los alfabetos asiáticos, griegos y rusos en el olvido, cada vez más se sustituye por el carácter extendido wchar_t, que codifica cada carácter con dos bytes (16 bits, es decir 65.536 combinaciones). En C/C++ es posible elegir el formato del carácter definitivamente por su tipo (char o wchar_t) o utilizando la macro TCHAR con una opción de compilación. De hecho, la macro TCHAR hace que el tipo carácter sea adaptable. En función de la directiva seleccionada, cada TCHAR se sustituye en la compilación por un char o por un wchat_t. La directiva en cuestión se ajusta en la **Página de propiedades** del proyecto C/C++, pestaña **General**, opción **Character Set**.



En .NET, el System.Char de manera nativa está en UNICODE, por lo tanto muy cerca del wchar_t del C/C++.

No hay que confundir el carácter C#, que es un alias de System.Char y que contiene el UNICODE, con el carácter del C/C++ que contiene el ANSI.

2. Codificación de las cadenas

En C/C++, una cadena es una serie de caracteres terminada con un 0. También este 0 sirve para delimitar las funciones de cálculo de longitud, visualización y manipulación de cadenas. En C# hemos visto que la cadena se encapsula en un objeto string, que protege la serie de caracteres y ofrece un conjunto de métodos y atributos para explotarla. Para poder pasar un objeto string C# a una cadena C/C++, se llama al Marshal. El siguiente ejemplo (proyecto PInvoke03 para descargar) muestra la transmisión de un objeto string a una DLL C/C++.

3. Transmisión de las cadenas

A continuación se muestra la parte C/C++ con una función muy sencilla que muestra la cadena que se le pasa como argumento. Esta versión funciona con cadenas de tipo UNICODE.

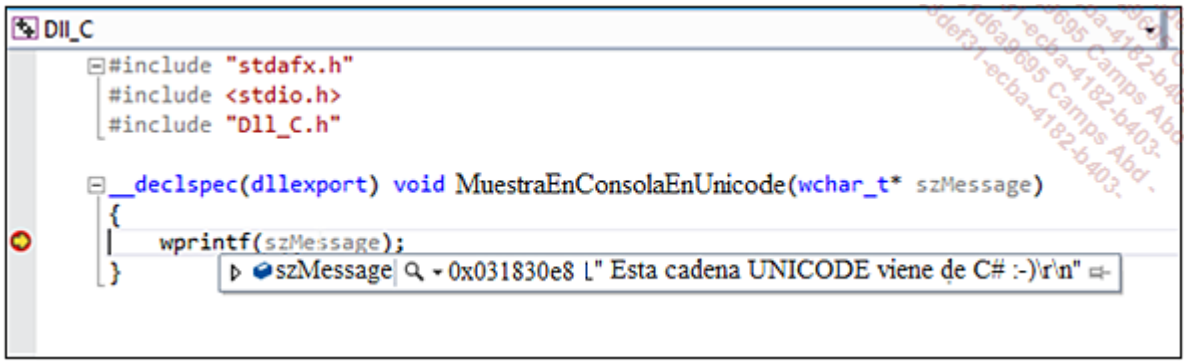
```
#include "stdafx.h"

#include <stdio.h>

#include "Dll_C.h"


__declspec(dllexport) void MuestraEnConsolaEnUnicode(wchar_t*

szMessage)

{

    wprintf(szMessage);

}
```

A continuación se muestra la parte C# que declara y utiliza la función C/C++:

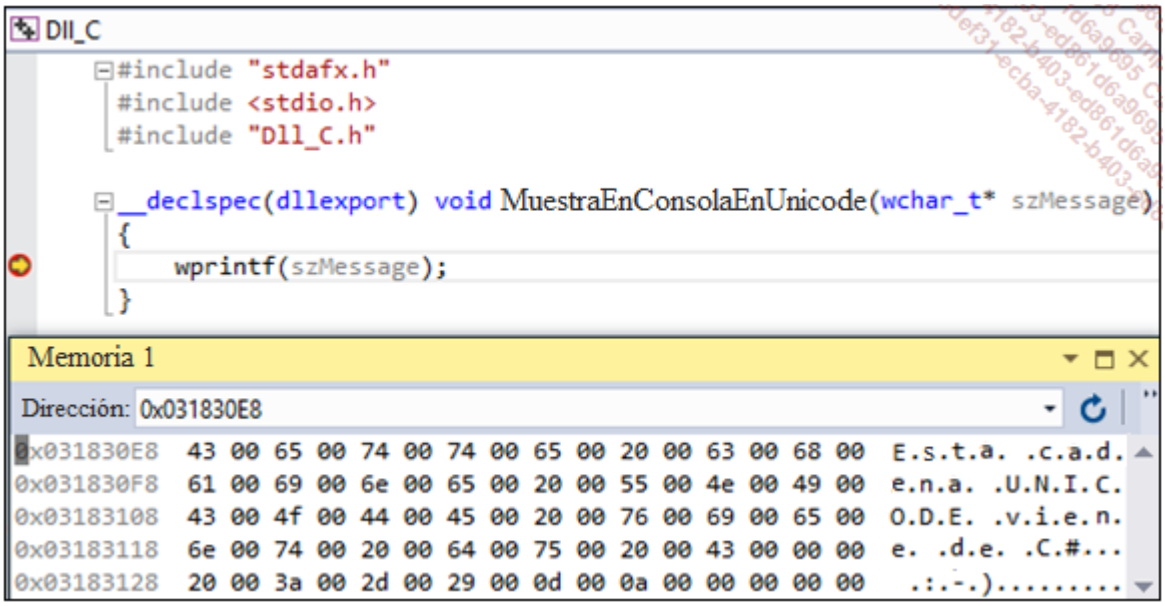
```
using System.Runtime.InteropServices;


namespace ConsoleApp
{
    class Program
    {
        [DllImport("Dll_C.dll", EntryPoint =
"MuestraEnConsolaEnUnicode", CallingConvention =
CallingConvention.Cdecl)]
        static extern void
MuestraEnConsolaEnUnicode([MarshalAs(UnManagedType.LPWStr)]string
szMessage);


        static void Main(string[] args)
        {
            MuestraEnConsolaEnUnicode("Esta cadena UNICODE viene
de C# :-)\r\n");
        }
    }
}
```

El objeto string se convierte en un puntero de wchat_t para el Marshal. Las reglas de conversión están en la directiva del Marshaling, que precede a la declaración del string szMessage: [MarshalAs(UnManagedType.LPWStr)]. Informa al sistema de que los datos contenidos en el objeto se deberán convertir en cadenas de caracteres UNICODE.

Poniendo un punto de ruptura en la función C/C++, se puede comprobar que la cadena está en formato correcto.



Podemos ir un poco más adelante y comprobar si un carácter está bien representado por dos bytes abriendo una ventana de memoria en szMessage.



Las funciones de las DLL que gestionan las cadenas de caracteres no funcionan necesariamente en UNICODE. Para pasar a modo MULTIBYTE/ANSI, es suficiente con modificar la directiva de Marshaling en [MarshalAs(UnManagedType.LPStr)].

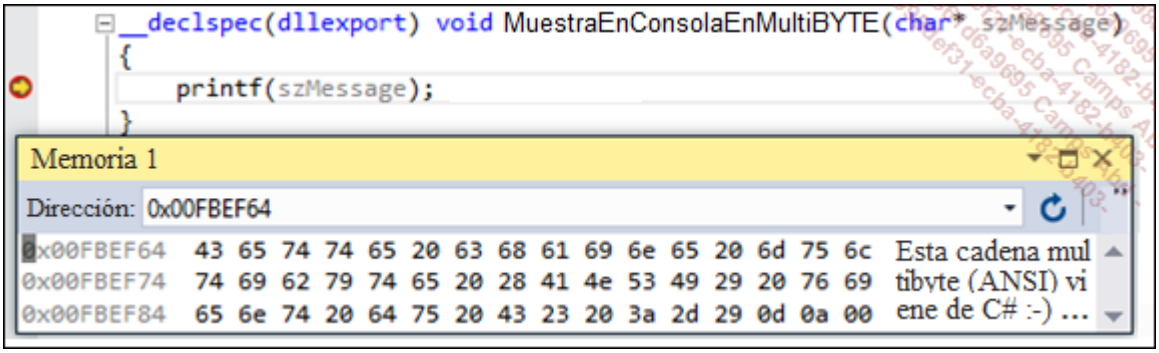
Por parte de la DLL C/C++ la función de visualización ANSI es:

```
#include "stdafx.h"
#include <stdio.h>
#include "Dll_C.h"


__declspec(dllexport) void MuestraEnConsolaEnMultibyte(char*
szMessage)
{
    printf(szMessage);
}
```

Por parte de C#:

```
using System.Runtime.InteropServices;


namespace ConsoleApp
{
    class Program
    {
        [DllImport("Dll_C.dll", EntryPoint =
"MuestraEnConsolaEnMultibyte", CallingConvention =
CallingConvention.Cdecl)]
        static extern void MuestraEnConsolaEnMultibyte([MarshalAs
(UnManagedType.LPStr)]string szMessage);


        static void Main(string[] args)
        {
            MuestraEnConsolaEnMultibyte("Esta cadena multibyte (ANSI)
viene de C# :-)\r\n");
        }
    }
}
```

Durante la ejecución de la función C/C++ se observa que los caracteres de la cadena recibida de C# están codificados con un byte.



Intercambio de tablas

1. De C# a C/C++

En C/C++, una tabla es un puntero a una zona de memoria que contiene un número definido de "casillas". Estas casillas están tipadas y su número se fija durante la creación de la tabla. Tenemos tablas de bytes, int, char, etc. En C#, la tabla es un objeto que también contiene una sucesión de datos, con un juego de métodos y atributos para explotarlos.

El siguiente ejemplo (proyecto PInvoke04 para descargar) va a mostrar cómo una tabla puede pasar de C# a C/C++.

Por parte de C/C++ hay una función que recupera una tabla y muestra su contenido en la consola. La tabla se declara por un puntero tipado. El tamaño dinámico se pasa como segundo argumento, lo que permite realizar una iteración correcta de la tabla.

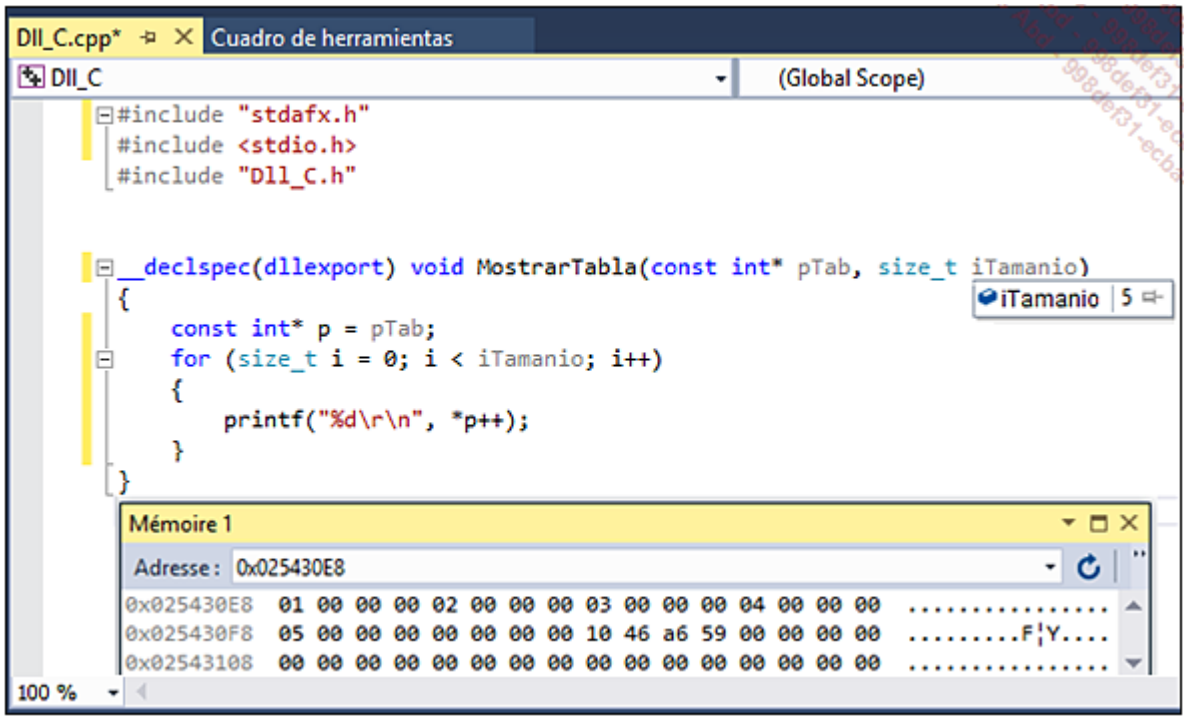
```
#include "stdafx.h"
#include <stdio.h>
#include "Dll_C.h"


__declspec(dllexport) void MostrarTabla(const int* pTab, size_t
iTamanio)
{
    int* p = pTab;
    for (size_t i = 0; i < iTamanio; i++)
    {
        printf("%d\r\n", *p++);
    }
}
```

Por parte de C#, la función C se declara con un primer argumento de tipo tabla precedido por la directiva [In]. Esta directiva [In] es el acceso directo de [InAttribute]; informa al Marshal de que la tabla va de C# **en** la función C. El tamaño de la tabla se declara a continuación como segundo argumento.

```
using System.Runtime.InteropServices;


namespace ConsoleApp
{
```

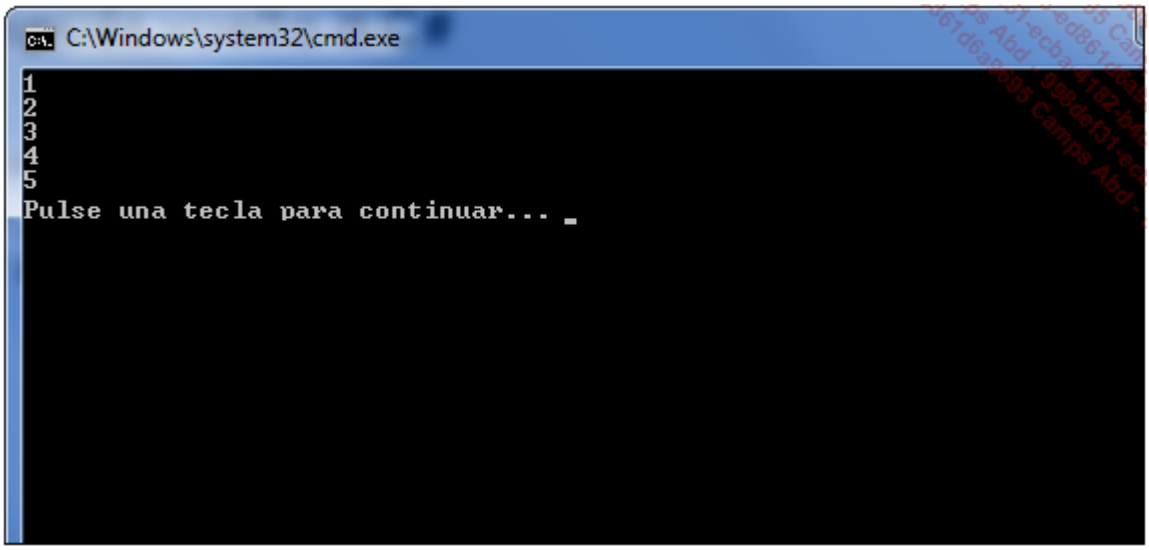
```
    class Program
    {
        [DllImport("Dll_C.dll", EntryPoint = "MostrarTabla",
CallingConvention = CallingConvention.Cdecl)]
        static extern void MostrarTabla([In] int[] tabDeInt, int
tamañoDeLaTablaInt);

        static void Main(string[] args)
        {
            int[] tabDeInt = new[] { 1, 2, 3, 4, 5 };
            MostrarTabla(tabDeInt, tabDeInt.Length);
        }
    }
}
```

Deteniendo la ejecución en la primera línea de la función C/C++, se puede comprobar el puntero y el número de entradas a tratar.



Para los desarrolladores C# que descubren los punteros, hay que observar que el puntero está tipado para recorrer una sucesión de enteros. Como un entero ocupa cuatro bytes,

entonces en cada incremento del puntero hace saltos de cuatro bytes. A esto se le llama aritmética de los punteros.



2. De C# a C/C++ y después vuelta a C#

El siguiente ejemplo continua la aventura, pasando de C# a C/C++ una tabla de caracteres escritos en minúsculas, que la función C/C++ va a convertir en mayúsculas, para devolverse finalmente a C#.

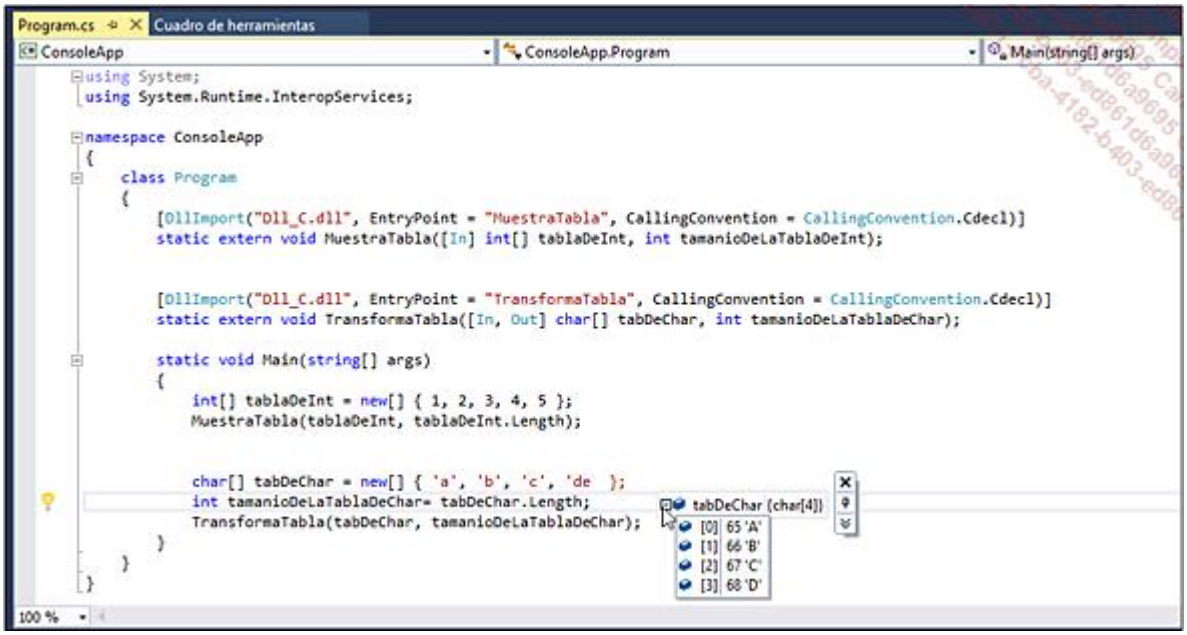
Por parte de C/C++:

```
#include "stdafx.h"
#include <stdio.h>
#include "Dll_C.h"

__declspec(dllexport) void TransformaTabla(char* pTab, size_t
iTamanio)
{
    char* p = pTab;
    for (size_t i = 0; i < iTamanio; i++)
    {
        *p += 'A'-'a';
        p++;
    }
}
```

Por parte de C#, la diferencia afecta al uso del atributo [In, Out] que prefija la tabla. Este atributo permite definir un viaje de "ida y vuelta " de la tabla.

```
using System;
using System.Runtime.InteropServices;

namespace ConsoleApp
{
    class Program
    {
        [DllImport("Dll_C.dll", EntryPoint = "TransformaTabla",
CallingConvention = CallingConvention.Cdecl)]
        static extern void TransformaTabla([In, Out] char[]
tabDeChar, int tamanioDeLaTablaDeChar);

        static void Main(string[] args)
        {
            char[] tabDeChar = new[] { ’a’, ’b’, ’c’, ’de  };
            int tamanioDeLaTablaDeChar = tabDeChar.Length;
            TransformaTabla(tabDeChar, tamanioDeLaTablaDeChar);

        }
    }
}
```

Un punto de ruptura en el código C# permite comprobar que se ha modificado correctamente la tabla de inicio de C# por la función C/C++.



Compartición de estructuras

Recordemos que el tipo structure reúne varios campos que pueden tener tipos diferentes. El tipo structure se utiliza mucho en los intercambios entre C# y C/C++, porque evita que las funciones tengan demasiados argumentos y parezca un conjunto de información de características del objeto a tratar.

El siguiente ejemplo (proyecto PInvoke05 para descargar) muestra la transmisión de una estructura que representa un cliente por su nombre, apellido, edad y número de teléfono.

1. Declaración de las estructuras

Por parte de C/C++:

```
#pragma pack(push, AppPack)
#pragma pack(1)

struct Cliente
{
    bool activo;
    char nombre[25];
    char apellido[25];
    BYTE edad;
    char telefono[20];
    int cuentaCliente;
};

#pragma pack(pop, AppPack)
```

Cada campo "texto" de la estructura se almacena en una tabla de caracteres de tamaño predefinido. Una directiva de "packing" permite fijar la alineación de los campos en el byte. En función de las optimizaciones del compilador C/C++, los campos de las estructuras no son necesariamente contiguos en memoria. El compilador puede insertar bytes entre ellos para que el microprocesador acceda más rápidamente, porque se optimizó para realizar "pasos" de ocho bytes, por ejemplo. Estos bytes "intermedios" añadidos por el compilador, no contienen datos, pero en el caso de un intercambio con C# provocarán desplazamientos en memoria.

Por este motivo se utiliza la directiva C/C++ de packing, para fijar la alineación independientemente de las optimizaciones y del tipo de compilación de la DLL. En nuestro ejemplo, la alineación se hace en el byte.

Por parte de C#:

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct Cliente
{
    [MarshalAs(UnManagedType.I1)]
    public bool activo;
    [MarshalAs(UnManagedType.ByValArray, SizeConst = 25)]
    public char[] nombre;
    [MarshalAs(UnManagedType.ByValArray, SizeConst = 25)]
    public char[] apellido;
    public byte edad;
    [MarshalAs(UnManagedType.ByValArray, SizeConst = 20)]
    public char[] telefono;
    public int cuentaCliente;
}
```

El atributo del primer campo de la estructura merece algunas explicaciones. El tipo booleano (bool) no existe como tal en C "puro". Se utiliza generalmente BOOL, que es un sinónimo del tipo int con sus dos valores FALSE/TRUE que son, de hecho, macros.

```
typedef int BOOL;
#ifndef FALSE
#define FALSE               0
#endif

#ifndef TRUE
#define TRUE                1
#endif
```

Por tanto, en C puro, el tipo BOOL es un entero que ocupa cuatro bytes en memoria y es por lo que el tipo bool de C# se marshala por defecto con cuatro bytes.

En la estructura Cliente C/C++, el campo activo es de tipo bool. Este tipo está permitido en la estructura C, porque el compilador es C++ y aporta algunas mejoras respecto a C, de las que el tipo bool forma parte. En C++, un booleano ocupa un byte, por tanto, si no se hace nada habrá un desplazamiento en la estructura.

El atributo [MarshalAs(UnManagedType.I1)] permite marshalar el campo "activo" en un byte.

En C#, la referencia de una tabla no puede asumir su tamaño, es necesario indicar a Marshal la dimensión que le deberá asignar durante los intercambios con C/C++. Es el objetivo del

atributo [MarshalAs(UnManagedType.ByValArray, SizeConst =25)], que indica que la tabla se debería transmitir elemento por elemento (y no por referencia), y que este número de elementos se fija a 25. La estructura también incluye un atributo que fija el orden de los argumentos y su alineación en el byte para estar en fase con la declaración de packing de C.

2. Utilización de las estructuras

Por parte de C/C++:

```
__declspec(dllexport) void RegistrarCliente(Cliente stCliente)
{
    printf("Cliente\r\nActivo: %d\r\nNombre: %s\r\nApellido:
%s\r\nEdad: %d\r\nTelefono: %s\r\n",
        stCliente.activo, stCliente.nombre, stCliente.apellido,
stCliente.edad, stCliente.telefono);
}
```

La función está en lenguaje C pero, al ser el entorno de desarrollo de tipo C++, consigue aligerar la declaración de uso de la estructura.

La función RegistrarCliente recupera una estructura de tipo Cliente y muestra todos los campos.

Por parte de C#:

```
[DllImport("Dll_C.dll", EntryPoint = "RegistrarCliente",
CallingConvention = CallingConvention.Cdecl)]
static extern void RegistrarCliente([In] Cliente cliente);
```

La declaración de la función C es clásica. Observe el atributo [In] que permite definir que la instancia de la estructura Cliente se va a pasar a la función C.

```
static void Main(string[] args)
{
    string nombre = "Ángel";
    string apellido = "Sánchez";
    byte edad = 42;
    string telefono = "9456587854";



    var cliente = new Cliente();
    cliente.activo = true;
    cliente.nombre = new char[25];
    cliente.apellido = new char[25];
    cliente.telefono = new char[20];

    Buffer.BlockCopy(nombre.ToCharArray(), 0,
        cliente.nombre, 0, nombre.Length * sizeof(char));
    Buffer.BlockCopy(apellido.ToCharArray(), 0,
        cliente.apellido, 0, apellido.Length * sizeof(char));
    cliente.edad = edad;
    Buffer.BlockCopy(telefono.ToCharArray(), 0,
        cliente.telefono, 0, telefono.Length * sizeof(char));

    RegistrarCliente(cliente);
}
```

Este código minimalista asigna y rellena una estructura de tipo Cliente. Observe la conversión entre un objeto string y una tabla de caracteres realizada con ayuda del método BlockCopy de la clase Buffer. El campo CuentaCliente no se utiliza por el momento sino más adelante.

[Ctrl][F5] permite comprobar el funcionamiento.

Ahora se va a recodificar un poco el código (proyecto PInvoke06 para descargar) para que el contenido de la estructura se pueda modificar por C/C++ y que esta modificación se devuelva a C#. El campo cuentaCliente se va a actualizar por la capa C/C++.

Por parte de C/C++, para que las modificaciones se añadan a la estructura original y no a una copia como es el caso del código anterior, es necesario que la estructura se pase a la función C por puntero (manera C "puro") o por referencia (manera C++).

A continuación se muestra el código por puntero C "puro":

```
__declspec(dllexport) void RegistrarCliente(Cliente* pStCliente)
{
    printf("Cliente\r\nActivo: %d\r\nNombre: %s\r\nApellido:
%s\r\nEdad: %d\r\nTelefono: %s\r\n",
        pStCliente->activo, pStCliente->nombre, pStCliente->apellido,
pStCliente->edad, pStCliente->telefono);

    pStCliente->cuentaCliente = 12345;
}
```

A continuación se muestra el código para el puntero C++:

```
__declspec(dllexport) void RegistrarCliente(Cliente& StCliente)
{
    printf("Cliente\r\nActivo: %d\r\nNombre: %s\r\nApellido:
%s\r\nEdad: %d\r\nTelefono: %s\r\n",
        StCliente.activo, StCliente.nombre, StCliente.apellido,
StCliente.edad, StCliente.telefono);
    StCliente.cuentaCliente = 12345;
}
```

Estos dos códigos tienen el mismo efecto. La segunda versión está un poco más optimizada.

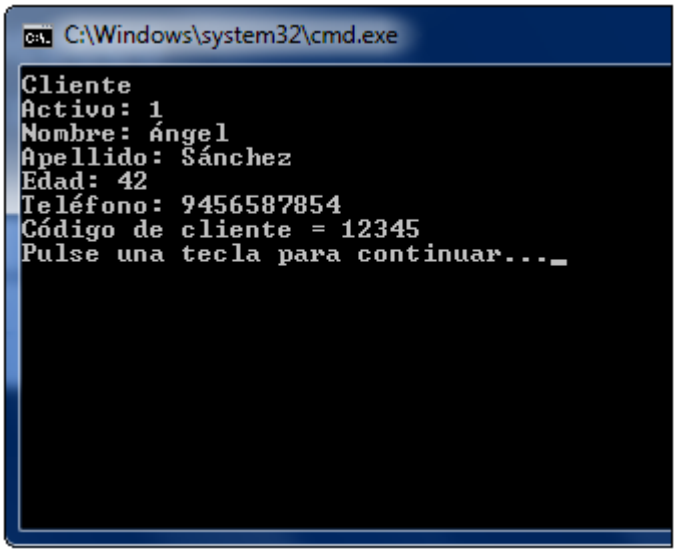
Por parte de C#, hay que declarar al Marshal que la estructura irá de C# a C/C++ y después de C/C++ a C#. La directiva [In,Out] es la que va a definir este comportamiento "P-Invoke", pero no hay que olvidarse de completar el prototipo C# añadiendo la palabra clave ref, indicando que el objeto se intercambiará por referencia.

```
[DllImport("Dll_C.dll", EntryPoint = "RegistrarCliente",
CallingConvention = CallingConvention.Cdecl)]
static extern void RegistrarCliente([In,Out] ref Cliente cliente);
```

A continuación se muestra su uso:

```
static void Main(string[] args)
{
    var cliente = new Cliente();
    cliente.activo = true;
    cliente.nombre = new char[25];
    cliente.apellido = new char[25];
    cliente.telefono = new char[20];

    Buffer.BlockCopy("Ángel".ToCharArray(), 0,
        cliente.nombre, 0, "Ángel".Length * sizeof(char));
    Buffer.BlockCopy("Sánchez".ToCharArray(), 0,
        cliente.apellido, 0, "Sánchez".Length * sizeof(char));
    cliente.edad = 25;
    Buffer.BlockCopy("9456587854".ToCharArray(), 0,
        cliente.telefono, 0, "9456587854".Length * sizeof(char));
    RegistrarCliente(ref cliente);

    Console.WriteLine(string.Format("Código de cliente = {0}",
cliente.cuentaCliente));

}
```

La ejecución muestra que el código de cliente se actualizó en C/C++.



Las directivas [In] y [Out]

En estos ejemplos hemos utilizado las directivas [In] y [Out] que simplifican de manera considerable la tarea, evitando codificación:

- Para [In]:
- Asignar una zona de intercambio.
- Copia con conversión "marshal" de la variable C# en la zona de intercambio, accesible por C/C++.

Para [Out]:

- Copia con conversión "marshal" de la variable C/C++, de la zona de intercambio a una variable C#.
- Para los dos:
- Desasignación de la zona de intercambio.

A continuación se muestra lo que sería la parte C# del último ejercicio (proyecto PInvoke07 para descargar), sin las directivas [In] y [Out]:

```
[DllImport("Dll_C.dll", EntryPoint = "RegistrarCliente",
CallingConvention = CallingConvention.Cdecl)]
static extern void RegistrarCliente(IntPtr cliente);

[StructLayout(LayoutKind.Sequential, Pack =1)]
public struct Cliente
{
    [MarshalAs(UnManagedType.I1)]
    public bool activo;
    [MarshalAs(UnManagedType.ByValArray, SizeConst =25)]
    public char[] nombre;
    [MarshalAs(UnManagedType.ByValArray, SizeConst =25)]
    public char[] apellido;
    public byte edad;
    [MarshalAs(UnManagedType.ByValArray, SizeConst =20)]
    public char[] telefono;
    public int cuentaCliente;
}


static void Main(string[] args)
{
    var cliente = new Cliente();
    cliente.activo = true;
    cliente.nombre = new char[25];
    cliente.apellido = new char[25];
    cliente.telefono = new char[20];

    Buffer.BlockCopy("Ángel".ToCharArray(), 0,
        cliente.nombre, 0, "Ángel".Length * sizeof(char));
    Buffer.BlockCopy("Sánchez".ToCharArray(), 0,
        cliente.apellido, 0, "Sánchez".Length * sizeof(char));
```

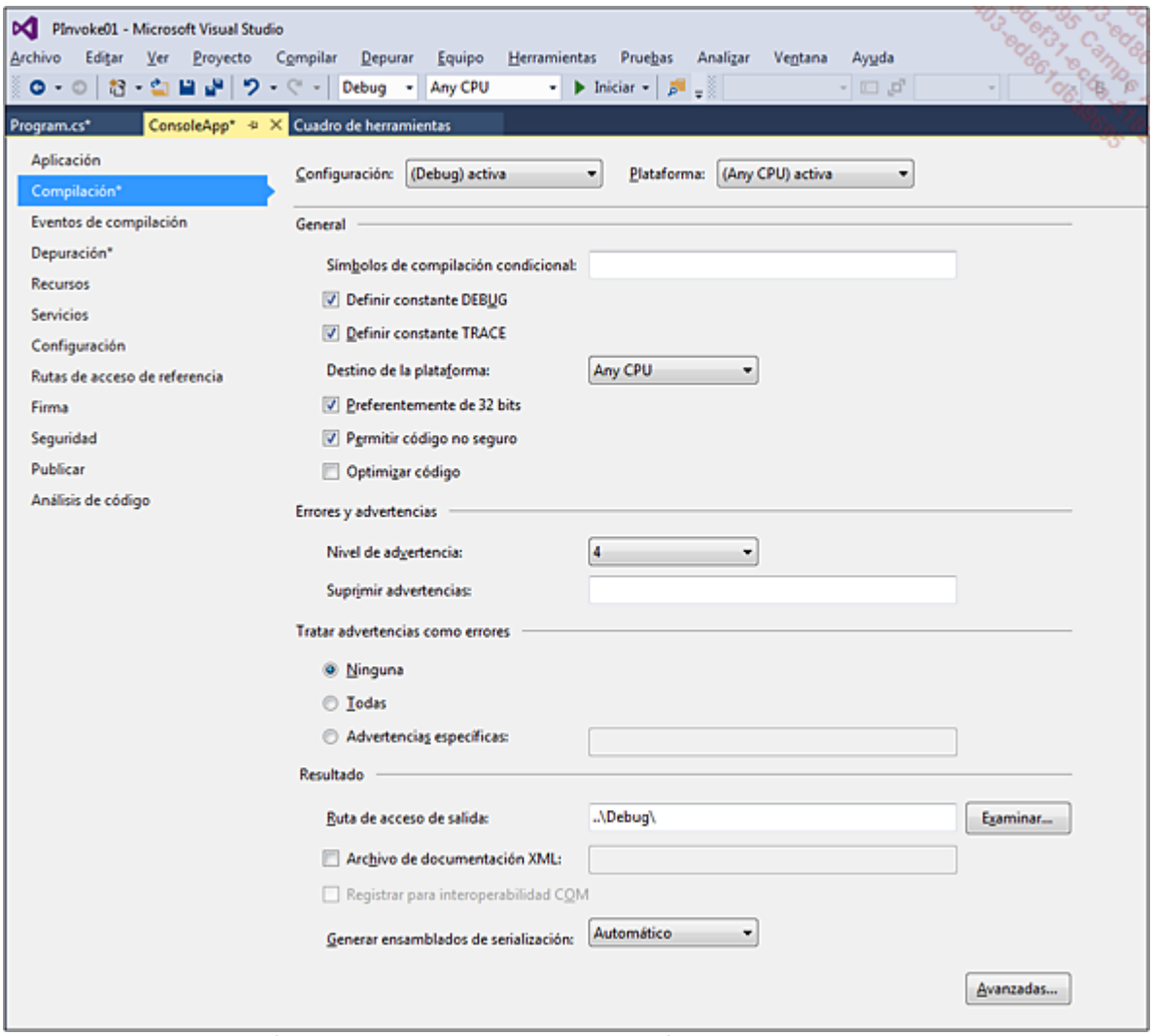
```
    cliente.edad = 25;
    Buffer.BlockCopy("9456587854".ToCharArray(), 0,
        cliente.telefono, 0, "9456587854".Length * sizeof(char));

    IntPtr pStCliente = Marshal.AllocHGlobal(Marshal.SizeOf(typeof(Cliente)));
    Marshal.StructureToPtr(cliente, pStCliente, false);



    RegistrarCliente(pStCliente);

    cliente = (Cliente)Marshal.PtrToStructure(pStCliente,
typeof(Cliente));

    Marshal.FreeHGlobal(pStCliente);

    Console.WriteLine(string.Format("Código de cliente = {0}",
cliente.cuentaCliente));

}
```

Las líneas en negrita representan los cambios respecto al ejemplo anterior.

Vemos en el prototipo de la función C el tipo IntPtr, que representa un puntero a memoria o incluso un handle clásico. Este puntero contiene la dirección de inicio de la zona de intercambio de memoria asignada por Marshal.AllocHGlobal y por tanto, el tamaño corresponde a la estructura Cliente. A continuación, Marshal.StructureToPtr realiza una copia "marshalada" del contenido de la estructura, es decir, teniendo en cuenta los eventuales atributos que prefijan cada campo. Una vez que se realiza esta copia se llama a la función C. Como retorno de la función C tenemos el procedimiento inverso, es decir, vuelve a copiar la zona de intercambio en la estructura Cliente.

Los desarrolladores de C/C++ tienen la costumbre de trabajar con punteros, pero los diseñadores de C# querían, por razones obvias, minimizar el uso de este tipo demasiado permisivo. Sin embargo, algunas veces puede ser útil visualizar el contenido de una zona de memoria, como por ejemplo la zona de intercambio C# C/C++ que acabamos de utilizar.

Para esto, hay que indicar al compilador que vamos a escribir un código "no seguro", utilizando como alias unsafe. Esto se configura en las propiedades del programa C# marcando la opción **Permitir código no seguro**, que está en la pestaña **Compilación**.

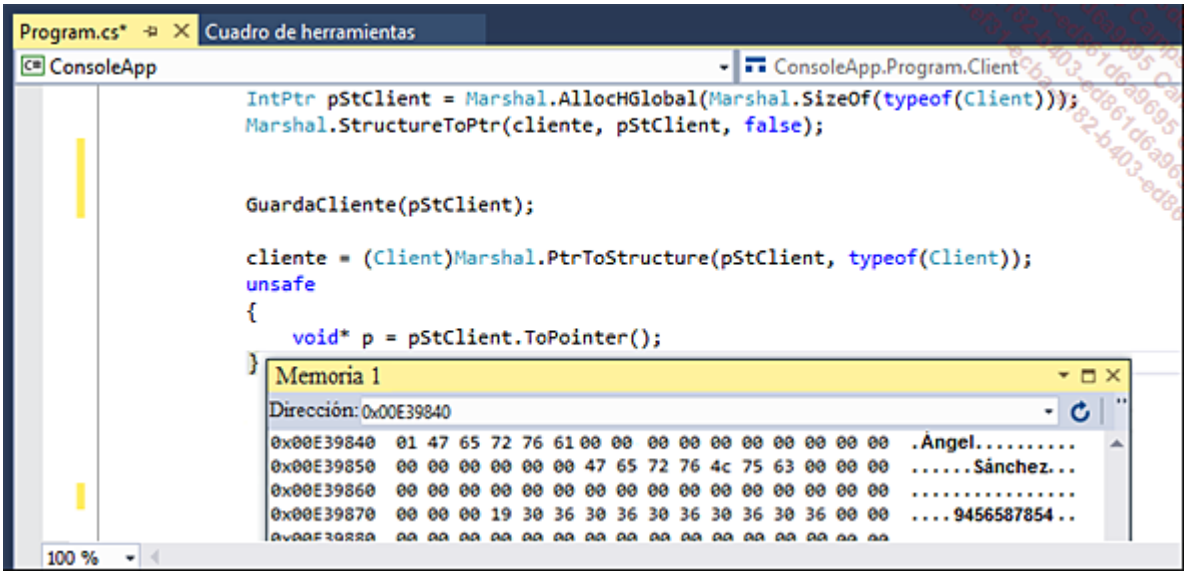


Obviamente esta opción se debe utilizar con moderación, porque la zona unsafe que vamos a definir ahora va a ejecutar el código que la CLR no podrá comprobar y, por lo tanto, podrá introducir riesgos de seguridad y estabilidad.

A continuación se muestra la inserción de la zona unsafe, que va a permitir recuperar y visualizar la zona de intercambio de actualización por la capa C/C++:

```
IntPtr pStCliente =

Marshal.AllocHGlobal(Marshal.SizeOf(typeof(Client)));

Marshal.StructureToPtr(cliente, pStClient, false);



GuardaCliente(pStClient);


cliente = (Client)Marshal.PtrToStructure(pStClient,

typeof(Client));

unsafe

{

    void* p = pStClient.ToPointer();

}


Marshal.FreeHGlobal(pStClient);
```

Deteniendo la ejecución en la llave cerrada de la zona unsafe es posible abrir una ventana de memoria y solicitar la visualización de la zona a la que apunta p.



Vemos los campos de la estructura "de principio a fin", porque la alineación se fijó al byte. Observe en la penúltima línea el código hexadecimal 0x39 0x30 0x00 0x00, que se corresponde con el código de cliente 12345 decimal convertido y colocado en memoria de manera "Intel" (little big endian).

Realización de un "wrapper"

En las secciones anteriores, la codificación del acceso a las funciones de las DLLs no es óptima. De hecho, la dirección de la función C/C++ se evalúa en cada llamada y no hay noción de instancia de usuario en la DLL. Para remediar esto, el desarrollador C# podrá crear una clase llamada "wrapper" que va a:

- gestionar las cargas, resolver las entradas y descargar la DLL no gestionada.
- proporcionar métodos C# que van a llamar a las funciones no gestionadas.

El wrapper puede ser objeto de un ensamblado "separado". Este generalmente es el caso para los periféricos programables. El fabricante pone a disposición de los desarrolladores un wrapper en forma de ensamblado, es decir una DLL, que basta con referenciar en el proyecto para poder utilizarla.

Por motivos de simplificación, el wrapper de nuestro ejemplo aparece en el proyecto C# en forma de una clase (proyecto PInvoke08 para descargar).

La DLL a wrapper es muy sencilla. Contiene dos funciones: una recupera y guarda un valor, mientras que la segunda lo restituye.

```
#include "stdafx.h"
#include <stdio.h>
#include "Dll_C.h"



static int DatoAGuardar = 0;



__declspec(dllexport) void Copia(int datoAGuardar)
{
    DatoAGuardar = datoAGuardar;
}


__declspec(dllexport) int Restituye()
{
    return DatoAGuardar;
}
```

1. Una región "NativeMethods"

La carga de la DLL y la resolución de sus puntos de entrada se va a realizar mediante la llamada a algunas funciones de la API Win32. Incluso si no es una obligación, es conveniente declarar estas funciones en una clase separada.

```
#region WIN32_DLL_MANAGER


public class Win32DllManager

{

    [DllImport("kernel32.dll", CharSet = CharSet.Unicode)]

    public static extern IntPtr LoadLibrary(string lpFileName);


    [DllImport("kernel32.dll")]

    public static extern bool FreeLibrary(IntPtr module);


    [DllImport("kernel32.dll", CharSet = CharSet.Unicode)]

    public static extern IntPtr GetProcAddress(IntPtr hModule,

[MarshalAs(UnManagedType.LPStr)]string procName);

}


#endregion
```

La función LoadLibrary carga una DLL, la función GetProcAddress permite recuperar un puntero sobre una de sus funciones a partir de su nombre y, por último, la función FreeLibrary permite descargar la DLL.

2. Almacenamiento de la información de la DLL nativa

La función LoadLibrary devuelve un "handle" sobre la DLL cargada. Este handle se almacenará en un atributo de tipo IntPtr, ya visto en el ejemplo anterior. Este handle se utilizará para resolver los puntos de entrada y para descargar la DLL al final de su uso.

El tipo delegate se adapta para recibir la dirección de un punto de entrada de la DLL. Hay que preceder su declaración con un atributo que indica su papel al Marshal.

[UnmanagedFunctionPointer(CallingConvention.Cdecl)]

UnmanagedFunctionPointer indica al Marshal que el delegate va a representar un puntero de función que contenga el código no gestionado. CallingConvention.Cdecl informa al Marshal del modo de transferencia de los argumentos de esta función.

A continuación se muestra en la clase Wrapper la zona de almacenamiento de la información de la DLL.

```
    public class Wrapper: IDisposable
    {
        #region Información de la DLL nativa

        private static IntPtr hModule = IntPtr.Zero;

        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        public delegate void _Copia(Int32 aGuardar);
        public static _CopiaSeguridad CopiaSeguridad = null;

        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        public delegate Int32 _Restituye();
        public static _Restituye Restituye = null;

        #endregion
    //...
}
```

3. Instanciación de DLL nativa

Ahora hay que codificar en el wrapper los métodos para inicializar el enlace con la DLL nativa y para liberarla con el código que la invoca o al final del ciclo de vida del objeto wrapper.

Es particularmente aconsejable implementar la interfaz IDisposable en un wrapper.

```
public class Wrapper: IDisposable
{
    #region Información de la DLL nativa

    private static IntPtr hModule = IntPtr.Zero;

    [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
    public delegate void _Copia(Int32 aGuardar);
    public static _CopiaSeguridad CopiaSeguridad = null;

    [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
    public delegate Int32 _Restituye();
    public static _Restituye Restituye = null;

    #endregion

    #region Ctor-Dtor Carga Descarga

    public Wrapper()
    {
        Carga();
    }

    public static bool IsLoaded { get { return hModule !=
IntPtr.Zero; } }

    public static bool Carga()
```

```
    {
        Descarga();
        bool bNoError = false;
        hModule = Win32DllManager.LoadLibrary("Dll_C.dll");
        if (hModule != IntPtr.Zero)
        {
            try
            {
                CopiaSeguridad
                = (_CopiaSeguridad)Marshal.GetDelegateForFunctionPointer(
                Win32DllManager.GetProcAddress(hModule,
"CopiaSeguridad"),
                typeof(_CopiaSeguridad));

                Restituye
                = (_Restituye)Marshal.GetDelegateForFunctionPointer(
                Win32DllManager.GetProcAddress(hModule,
"Restituye"),
                typeof(_Restituye));

                bNoError = true;

            }
            catch (Exception)
            {
                Descarga();
            }
        }
        return bNoError;
    }

    public static void Descarga()
    {
```

```
        if (IsLoaded)
        {
            Win32DllManager.FreeLibrary(hModule);
            hModule = IntPtr.Zero;
        }
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    ~Wrapper()
    {
        Dispose(false);
    }

    protected virtual void Dispose(bool
bCleanUpManagedAndNativeResources)
    {
        if (bCleanUpManagedAndNativeResources == true)
        {
            Descarga();
        }
    }
    #endregion
}
```

El método Carga utiliza los métodos de tipo static de nuestra clase Win32DllManager para cargar la DLL, recuperar sus puntos de entrada y almacenarlos en los delegate que se le reservan. Observe que el retorno de la función GetProcAddress pasa por una conversión de puntero de función hacia delegate antes del almacenamiento.

El método Descarga libera la DLL gestionada.

La mecánica del IDisposable permite descargar la DLL al final del ciclo de vida del wrapper, cuando el desarrollador que utiliza la clase haya olvidado llamar a Descarga.

4. Métodos de uso de la DLL gestionada desde el wrapper

Ahora que el wrapper ofrece métodos para cargar y descargar la DLL gestionada, se pueden crear métodos de explotación de esta DLL.

```
#region Métodos de explotación
public bool CsCopia(int aGuardar)
{
    if (!IsLoaded)
        Carga();
    if (!IsLoaded)
        return false;
    Copia(aGuardar);
    return true;
}

public bool CsRestituye(out int valor)
{
    valor = -1;
    if (!IsLoaded)
        return false;
    valor= Restituye();
    return true;
}
#endregion
```

5. Uso del wrapper

El uso de la clase Wrapper en C# es muy sencilla: basta con instanciarlo, llamar a su método de carga y después utilizar los métodos de explotación.

```
class Program
{
    static void Main(string[] args)
    {
        var miWrapper = new Wrapper();
        if (Wrapper.Carga())
        {
            miWrapper.CsCopia(12345);

            int valor = -1;
            miWrapper.CsRestituye(out valor);

            Wrapper.Descarga();
        }
    }
}
```

```
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            var miWrapper = new Wrapper();
            if (Wrapper.Carga())
            {
                miWrapper.CsCopia(12345);

                int valor = -1;
                miWrapper.CsRestituye(out valor);

                Wrapper.Descarga();

            }
        }
}
```

La ejecución de este fragmento de código, muestra que el valor 12345 se conservó entre las dos llamadas por la DLL C/C++.

Si no hemos utilizado el wrapper, la variable valor habría contenido -1, porque cada llamada a las funciones C/C++ fue objeto de una nueva instancia y, por tanto, reinicializa la memoria.

La siguiente imagen muestra una codificación "clásica":

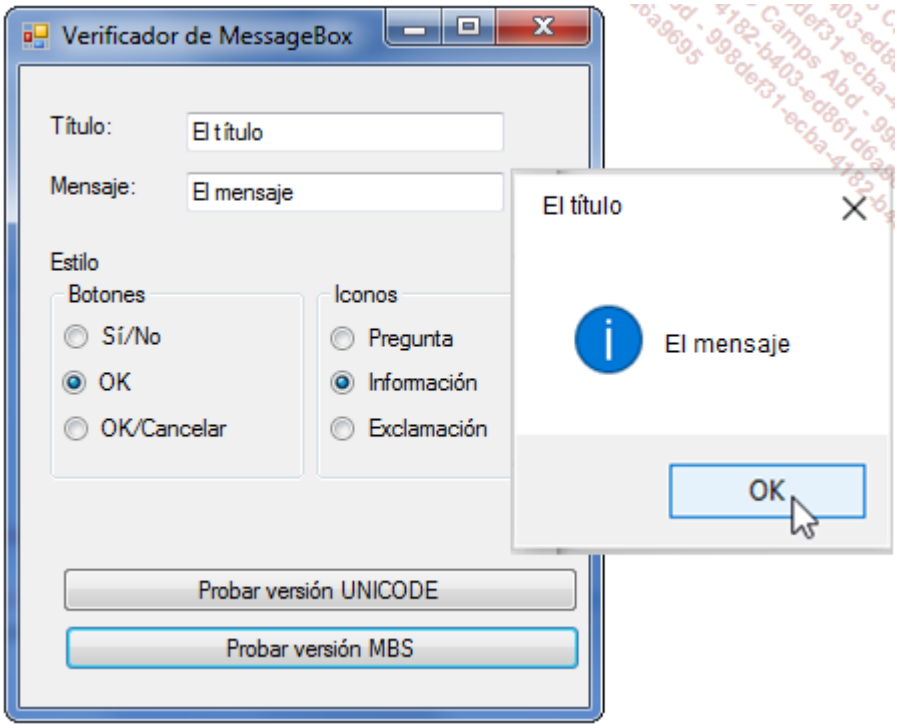
```
namespace ConsoleApp
{
    class Program
    {
        [DllImport("Dll_C.dll", EntryPoint = "CopiaSeguridad", CallingConvention = CallingConvention.Cdecl)]
        static extern CopiaSeguridad(int valor);
        [DllImport("Dll_C.dll", EntryPoint = "Restaurar", CallingConvention = CallingConvention.Cdecl)]
        static extern void Restaurar([Out] out int valor);
        static void Main(string[] args)
        {
            CopiaSeguridad(12345);
            int valor = -1;
            Restaurar(out valor);
        }

    }
}
```

valor -1

Ejercicio

1. Enunciado

El ejercicio propuesto consiste en realizar una aplicación gráfica que permite comprobar parcialmente la función MessageBox "nativa" de Windows. La interfaz a desarrollar se podría parecer a la siguiente:



La documentación de MessageBox está disponible en el sitio Web MSDN de Microsoft en la siguiente dirección: https://msdn.microsoft.com/es-es/library/windows/desktop/ms645505(v=vs.85).aspx

Se puede leer en qué DLL está la función, que esta función existe en los formatos Unicode y MBS y que combina con un determinado número de propiedades para realizar visualizaciones específicas.

2. Corrección

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Runtime.InteropServices;
using System.Windows.Forms;
namespace WindowsFormsApplications
{
    public partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
            radioButtonInformacion.Checked = true;
            radioButtonOk.Checked = true;
            textBoxTitulo.Text = "El título";
            textBoxMensaje.Text = "El Mensaje";
        }
        [DllImport("User32.dll", EntryPoint = "MessageBoxW",
CallingConvention = CallingConvention.Winapi)]
        static extern int MessageBoxW(IntPtr hWnd,
[MarshalAs(UnManagedType.LPWStr)]string text,
[MarshalAs(UnManagedType.LPWStr)]string titulo, int tipo);

        [DllImport("User32.dll", EntryPoint = "MessageBoxA",
CallingConvention = CallingConvention.Winapi)]
        static extern int MessageBoxA(IntPtr hWnd,
[MarshalAs(UnManagedType.LPStr)]string text,
[MarshalAs(UnManagedType.LPStr)]string titulo, int tipo);

      private void IntentarVersionUnicode(object sender, EventArgs e)
```

```
        {
            MessageBoxW(IntPtr.Zero, textBoxMensaje.Text,
textBoxTitulo.Text, getType());
        }

        private void IntentarVersionAnsi(object sender, EventArgs e)
        {
            MessageBoxA(IntPtr.Zero, textBoxMensaje.Text,
textBoxTitulo.Text, getType());
        }
        private int getType()
        {
            return getButtonType() | getIconType();
        }
        private int getButtonType()
        {
            if (radioButtonOkCancel.Checked == true)
                return 1;
            if (radioButtonSiNo.Checked == true)
                return 4;
            return 0;
        }
        private int getIconType()
        {
            if (radioButtonExclamacion.Checked == true)
                return 0x30;
            if (radioButtonPregunta.Checked == true)
                return 0x20;
            return 0x40;
        }
    }
}
```

Es la DLL User32.DLL la que alberga las dos versiones de MessageBox. La pequeña dificultad de este ejercicio era definir correctamente el Marshaling de las cadenas a mostrar en función de la versión de MessageBox utilizada. La corrección se proporciona en el proyecto PInvoke09-Exo, disponible para su descarga.

Introducción

Cuando un cliente pide un desarrollo, se elaboran las especificaciones, que describen las funcionalidades de alto nivel, en forma de casos concretos de uso. Este documento servirá más tarde para validar los desarrollos realizados. Estos casos de uso van a originar muchos objetos desarrollados por el equipo. Estos objetos se van a comunicar entre ellos con los métodos y las secuencias diseñados durante el análisis. La mayor parte de las veces, cada intercambio se realizará con los argumentos de "ida" y "vuelta". Los rangos admisibles de estos argumentos serán conocidos y estos objetos funcionarán generalmente bien, cuando reciban lo que esperan, en el momento que lo esperan. Pero, ¿qué sucede cuando aumente el ritmo o los argumentos que se pasan estén fuera de los límites?

Es ahí cuando se muestra la solidez de aplicación, en los casos extremos por operaciones adaptadas a los fallos y una buena protección de los datos. Para ganar este grado de fiabilidad, en primer lugar, cada eslabón de la cadena debe permanecer estable, sean cuales sean sus condiciones de explotación. Para ello hay que experimentar llevándolos a sus límites. El desarrollador deberá imaginar los peores casos de uso de sus objetos, desde su codificación, y observarlos inmediatamente generando el código más seguro posible. Para jugar en estos escenarios difíciles, escribirá una serie de pruebas llamadas pruebas **unitarias**, que validarán el correcto comportamiento de cada objeto y verificarán la mayor parte de código posible.

Más adelante, los objetos se van a relacionar e intercambiar datos. Las pruebas que validan estos intercambios inter-objetos se llaman **pruebas de integración**. En la medida de lo posible, el desarrollador prepara sus pruebas de integración escribiendo objetos ficticios que tienen las mismas propiedades que los futuros componentes reales. Esta etapa permite comprobar que las comunicaciones previstas están correctamente implementadas, incluso si los objetos ficticios no hacen ninguna otra operación, salvo algunas trazas.

Cuando todos los objetos están validados y se comunican correctamente entre ellos, se puede pasar a las **pruebas de comprobación** para controlar la aplicación completa, haciendo referencia a las especificaciones funcionales del cliente.

Probar es una tarea ingrata que hay que intentar automatizar lo máximo posible para que se pueda ejecutar a voluntad. Las pruebas se deben escribir con sentido y deben tener en cuenta la mayor parte posible de casos de uso.

El coste de un bug detectado y arreglado durante la fase de desarrollo, es ínfimo en relación con el que tendría si se detecta en producción.

La "testabilidad" debe formar parte de las restricciones durante el análisis del objeto. Teóricamente, cada módulo se debe poder probar de manera autónoma, lo que implica que los objetos estén débilmente acoplados.

En un mundo ideal, las pruebas unitarias solo deberán afectar a los métodos que solo realizan una única función, sin utilizar otros métodos.

Estas pruebas, antes muy secundarias, ahora son prioritarias y determinados métodos de desarrollo empiezan definiéndolas antes incluso de escribir las clases que se deben comprobar.

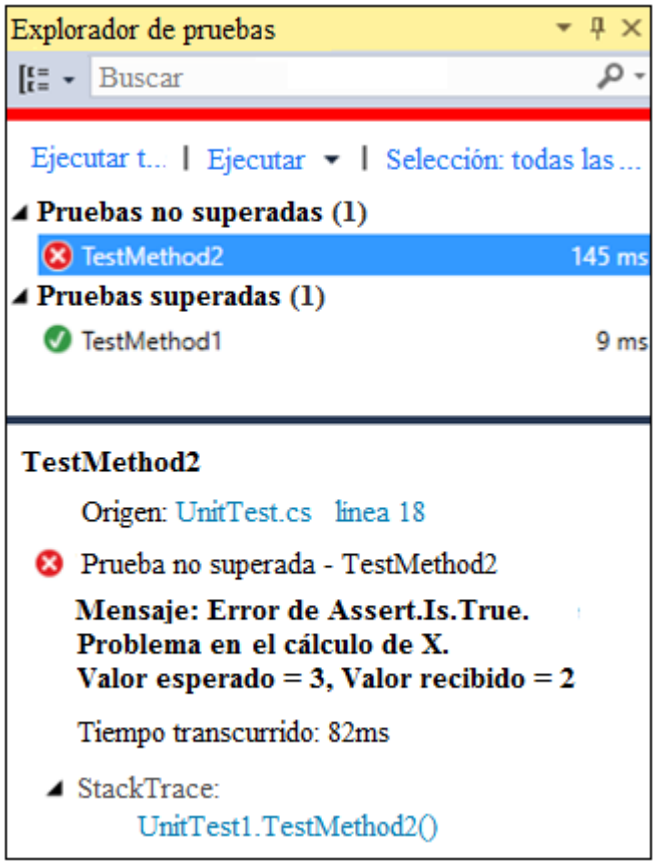
Esté desarrollando o en fase de mantenimiento, es muy tranquilizador saber que el conjunto de pruebas escritas para sus clases se ejecutan con éxito.

Entorno de ejecución de las pruebas unitarias

Siempre es posible escribir "pequeñas aplicaciones" autónomas que permitan probar los objetos de la futura "gran aplicación". Por ejemplo, un código cargado en una consola podrá instanciar la clase a probar, después desarrollar una serie de llamadas que muestren mensajes de error o que escriban los resultados en un archivo. Por supuesto es posible, pero no muy práctico.

.NET y Visual Studio 2015 son versiones que simplifican la codificación, ejecución y análisis de las pruebas unitarias. No son necesarias "pequeñas aplicaciones" autónomas; Visual Studio ofrece un tipo de proyecto especial que permite escribir **directamente** un conjunto de pruebas que el desarrollador podrá reproducir en su totalidad, en grupo (playlist) o individualmente, gracias al explorador de pruebas. Incluso será posible programar su ejecución automática después de la compilación. Los resultados de las pruebas se resumen en una tabla que utiliza los colores rojo y verde, y que permiten ir rápidamente en caso de error a la línea de código que ha fallado. Es posible ejecutar las pruebas en modo **Debug** y, por tanto, trazar los métodos llamados en los objetos que se están probando.

A continuación se muestra un ejemplo de resultado de ejecución de dos pruebas.



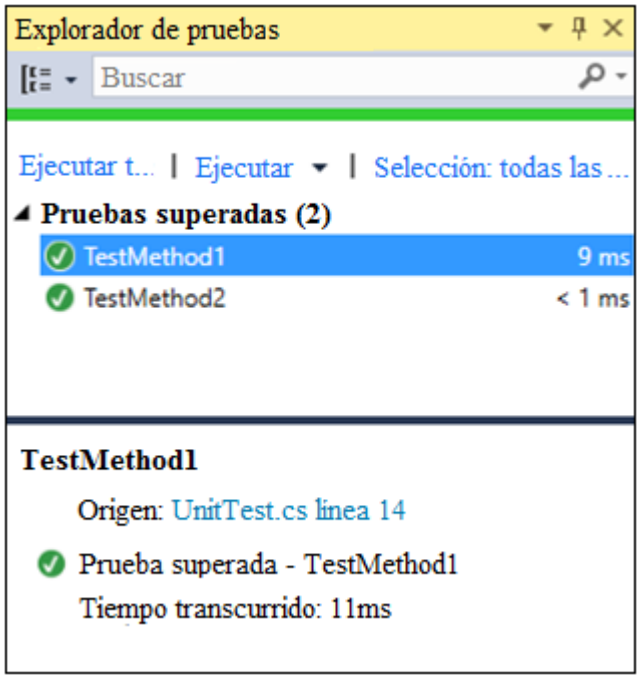
La parte superior de la pantalla muestra una línea roja gruesa que indica que al menos una prueba se ha desarrollado de manera errónea: TestMethod2 falló.

La parte inferior de la pantalla contiene la siguiente información adicional:

- un enlace de texto en la línea de inicio de la prueba,
- una explicación de la naturaleza del problema,
- el tiempo de ejecución de la prueba hasta el error,
- un enlace de hipertexto sobre la línea que presenta un problema en la prueba.

TestMethod1 ha tenido éxito. Aparece precedida de una marca verde. El explorador de pruebas también muestra su tiempo de ejecución de 9 ms.

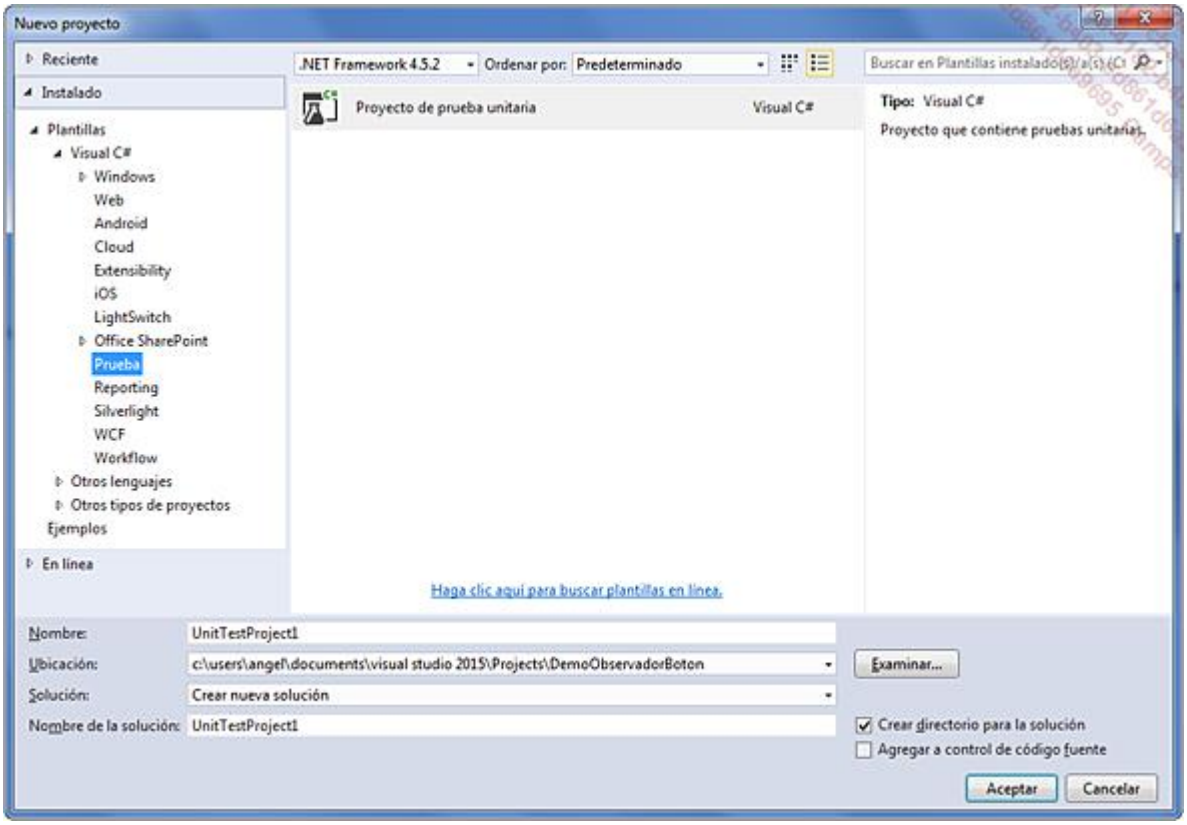
A continuación se muestra ahora un segundo ejemplo de resultados de ejecución de dos pruebas con éxito:



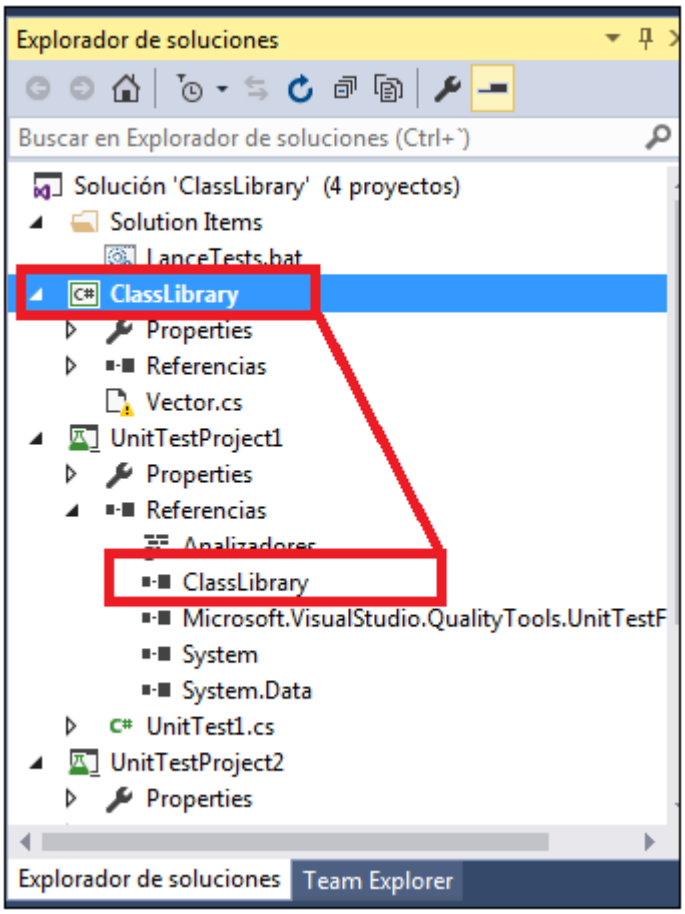
En la parte superior de la pantalla aparece un trazo verde grueso que significa que todas las pruebas se han desarrollado con éxito.

El proyecto de pruebas unitarias

Las pruebas son métodos de clase que forman parte de un proyecto especial Proyecto de prueba unitaria que añadimos, la mayor parte de las veces, a la solución que contiene las clases a probar.



Después hay que añadir al proyecto de prueba unitaria la o las referencias a los ensamblados que contienen las clases a probar.



La clase de pruebas

El código de inicio de una clase de prueba es el siguiente:

```
using System;
using Microsoft.VisualStudio.TestTools.TestClassing;
using ClassLibrary;

namespace TestClassProject1
{
    [TestClass]
    public class TestClass1
    {
        [TestMethod]
        public void PruebaMetodo1()
        {
        }
    }
}
```

Encontramos un espacio de nombres que contiene la clase TestClass1, que a su vez contiene el método PruebaMetodo1.

El nombre de la clase está precedido por el atributo TestClass, que permite al motor de ejecución identificarla como una clase de pruebas. Esta clase está cerrada, pero es posible definirla parcialmente (partial) y, por tanto, implementarla en varios archivos de código fuente.

Cada método de pruebas debe estar precedido por el atributo TestMethod, que la distinguirá de otros posibles métodos de la misma clase.

Contenido de un método de prueba

Cuando una prueba se ejecuta, la carga del ensamblado de destino se hace automáticamente además del entorno. La prueba debe instanciar la clase de destino si no es de tipo static, llamar a uno de sus métodos y validar el comportamiento esperado. Normalmente, solo debe haber una acción sobre el objeto para probar.

Por ejemplo, para un método Sumar, la prueba verifica que si 2 y 3 se pasan como argumentos, el resultado devuelto será 5.

Las comprobaciones utilizan los servicios de la clase Microsoft.VisualStudio.TestTools.TestClassing.Assert, que ofrece una colección de métodos mucho más completa que la de System.Diagnostics.Debug utilizada en este libro.

Entre este juego de métodos está el método IsTrue.

```
public static void IsTrue(bool condición, string mensaje);
```

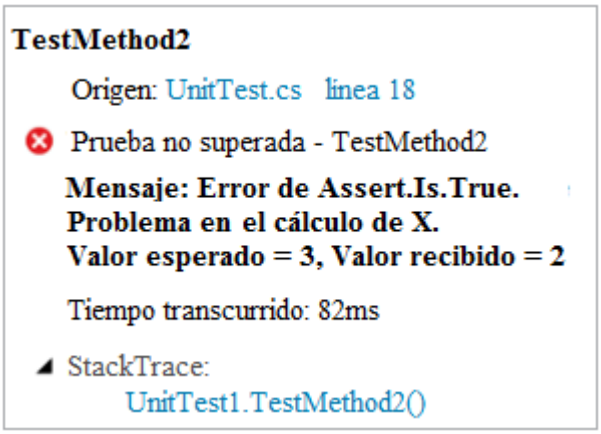
Este método permite comprobar que una condición es verdadera y, si no lo es, devolver un mensaje en el explorador de pruebas.

Algunos métodos de la clase Assert ofrecen una version con mensaje y una version sin mensaje. Es aconsejable utilizar la version con mensaje para que el análisis de los problemas sea mucho más intuitivo.

Ejemplo de uso de la version Assert.IsTrue con mensaje:

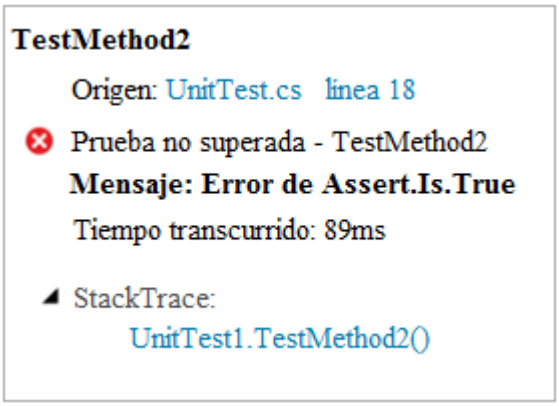
```
[TestMethod]
public void PruebaMetodo2()
{
    var vector = new Vector();
    vector.Agregar(new Vector() { X = 2, Y = 2 });
    Assert.IsTrue(vector.X == 3, "Problema en el cálculo de X
Valor esperado=3 Valor recibido=" + vector.X);
}
```

Resultado en el explorador de pruebas:



Ejemplo de uso de la version Assert.IsTrue sin mensaje:

```
[TestMethod]
public void PruebaMetodo2()
{
    var vector = new Vector();
    vector.Agregar(new Vector() { X = 2, Y = 2 });
    Assert.IsTrue(vector.X == 3);
}
```



La clase Assert contiene otros métodos de tipo static, por lo que a continuación se muestra una lista no exhaustiva:

AreEqual AreNotEqual	Compara dos valores o dos objetos. En el caso de objetos que hayan redefinido el método Equal, este se utilizará automáticamente.
AreSame AreNotSame	Compara las referencias de dos objetos.
IsNull IsNotNull	Compara una referencia con NULL.
IsInstanceOfTipo IsNotInstanceOfTipo	Comprueba si un objeto es de un tipo concreto.
IsTrue IsFalse	Comprueba una condición.

Una prueba también puede comprobar que no se genera ninguna excepción durante la ejecución de un método o, por el contrario, comprobar que se genera una excepción particular, como en la siguiente prueba:

```
[TestMethod]
[ExpectedException(typeof(System.NullReferenceException))]
public void PruebaMetodo3()
{
    var vector = new Vector();
    Vector vector2 = null;
    vector.Agregar(vector2);
}
```

El atributo añadido ExpectedException, seguido del tipo de excepción esperado, permite validar la prueba sobre esta excepción generada.

Los fragmentos de código anteriores forman parte de la solución ClassLibrary, disponible para su descarga.

Tratamientos de preparación y limpieza

La ejecución de las pruebas puede estar precedida por operaciones de inicialización y seguida de operaciones de "limpieza". Estas operaciones opcionales se escriben en métodos completados por atributos especiales que definen el momento en el que se ejecutarán durante el script.

Atributo del método	Cuándo se ejecutará el método
[AssemblyInitialize]	Una vez en cada inicio de la serie de pruebas de todas las clases.
[ClassInitialize]	Una vez en cada inicio de la serie de pruebas de la clase.
[TestInitialize]	Antes de cada prueba.
[TestCleanup]	Después de cada prueba.
[ClassCleanup]	Una vez al final de la ejecución de la serie de pruebas, de la clase.
[AssemblyCleanup]	Una vez al final de la ejecución de la serie de pruebas, de todas las clases.

Extracto de código que muestra la sintaxis de todos los métodos de inicialización:

```
[TestClass]
public class TestClass1
{
    [AssemblyInitialize]
    public static void MiAssemblyInitialize(TestContext tc)
    {
        Trace.WriteLine("Inicialización del assembly de las pruebas");
    }

    [AssemblyCleanup]
    public static void MiAssemblyCleanup()
    {
        Trace.WriteLine("Limpieza del assembly de las pruebas");
    }

    [ClassInitialize]
    public static void MiClaseInit(TestContext tc)
    {
        Trace.WriteLine("Inicialización de la clase TestClass1");
    }

    public TestClass1()
    {
        Trace.WriteLine("Constructor de TestClass1");
    }

    [ClassCleanup]
    public static void MiClaseCleanup()
    {
        Trace.WriteLine("Limpieza de la clase TestClass1");
    }
```

```
    [TestInitialize]
    public void MiPruebaInitialize()
    {
        Trace.WriteLine("Preparación antes de la prueba");
    }

    [TestCleanup]
    public void MiPruebaCleanup()
    {
        Trace.WriteLine("Limpieza después de la prueba");
    }

    [TestMethod]
    public void MiPrueba1()
    {
        Trace.WriteLine("Ejecución de Prueba1");
    }

    [TestMethod]
    public void MiPrueba2()
    {
        Trace.WriteLine("Ejecución de Prueba2");
    }
}
```

A continuación se muestra el resultado de la ejecución en modo Debug, capturado en la ventana Salida del código anterior:

```
Inicialización del assembly de las pruebas
Constructor de TestClass1
Inicialización de la clase TestClass1
Preparación antes de la prueba
Ejecución de Prueba1
Limpieza después de la prueba
Constructor de TestClass1
Preparación antes de la prueba
Ejecución de Prueba2
Limpieza después de la prueba
Limpieza de la clase TestClass1
Limpieza del assembly de las pruebas
```

Los fragmentos de código anteriores forman parte de la solución ClassLibrary, disponible para su descarga.

Los métodos que tratan del assembly y de la clase son de tipo static y reciben como argumento un objeto de tipo TestContext, que se presenta más adelante. El uso de los métodos que tratan del assembly es muy raro.

El método de inicialización de la clase es práctico para preparar los recursos que se utilizarán en las pruebas. El ejemplo clásico es la apertura de una conexión a una base de datos. El método de limpieza de la clase siempre se llamará de manera independiente, sea cuál sea el resultado de las pruebas. Para retomar el caso anterior, contendrá el cierre de la conexión a la base de datos. Preste atención, estos dos métodos son de tipo static, por lo que solo pueden actuar sobre objetos de tipo static.

Los métodos de inicialización y limpieza de las pruebas son de tipo dinámico y se ejecutan antes y después cada prueba. Hay un código que permite garantizar que todas las pruebas empiezan en las mismas condiciones.

Como muestra la captura anterior, la clase se instancia antes de cada prueba, provocando que se pase de manera obligatoria en los constructores de todos sus datos miembros.

TestContext y fuente de datos

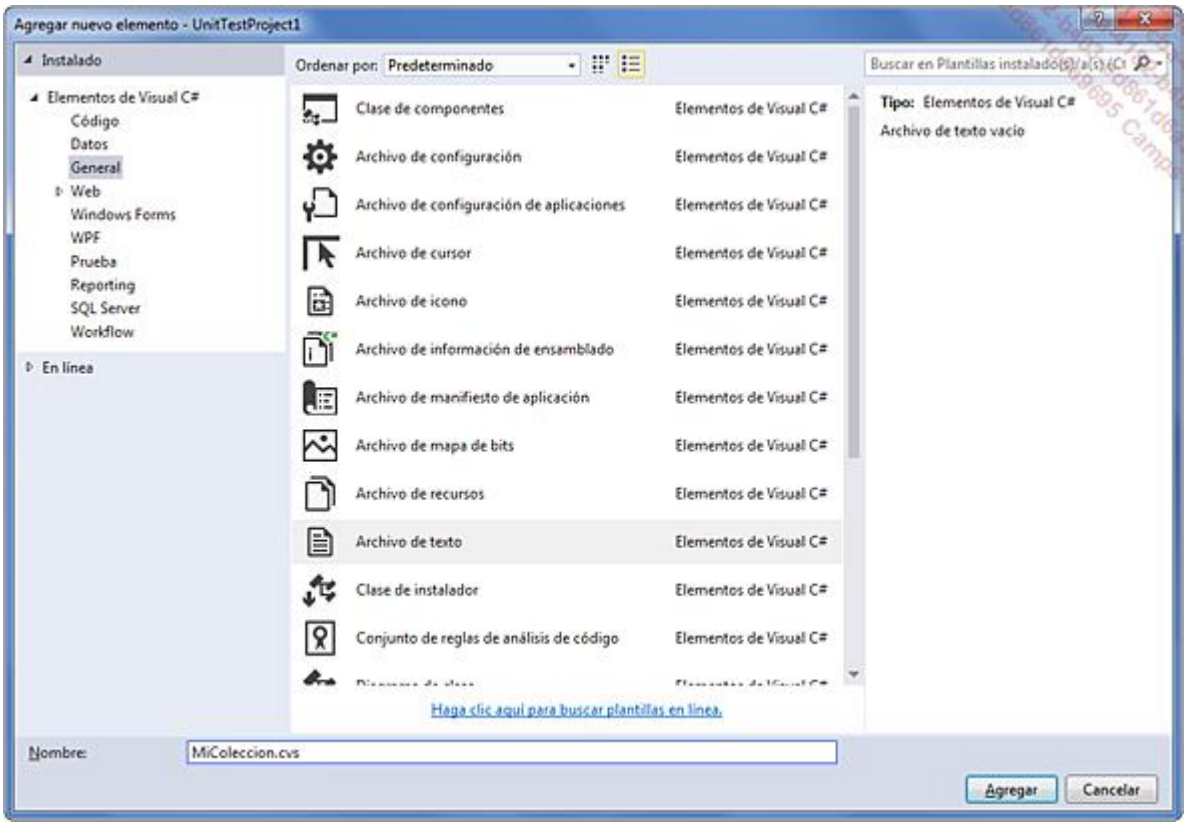
El método de inicialización de la clase recibe como argumento un objeto de tipo TestContext. El uso de este objeto es opcional. Contiene determinada información sobre la configuración de la prueba que se está ejecutando, pero sobre todo permite definir una fuente de datos que va a alimentar las pruebas. Hablamos de "data-driven unit tests" y se entiende rápidamente el interés de este modo de pruebas.

Imaginemos que vamos a comprobar el comportamiento de un método en cientos de casos de uso. La colección de información será difícil de escribir y mantener en el código. Cualquier cambio hará necesaria una re-compilación de la prueba. Por estas razones se externaliza la colección. El desarrollador puede gestionar la carga y posterior iteración en la colección. Sin embargo, este no es el objetivo de la prueba.

El framework ofrece la posibilidad de alimentar las pruebas con información que proviene de bases de datos, como SQL, XML o CSV. La persona que prueba debe definir el tipo y los argumentos de acceso a la base de datos y TestUnit se encarga de hacer el resto, llamando a la prueba implicada para cada registro de la tabla.

La siguiente demostración utiliza una colección en formato CSV (Comma-Separated Values).

En primer lugar, hay que añadir el archivo CSV al proyecto de pruebas.

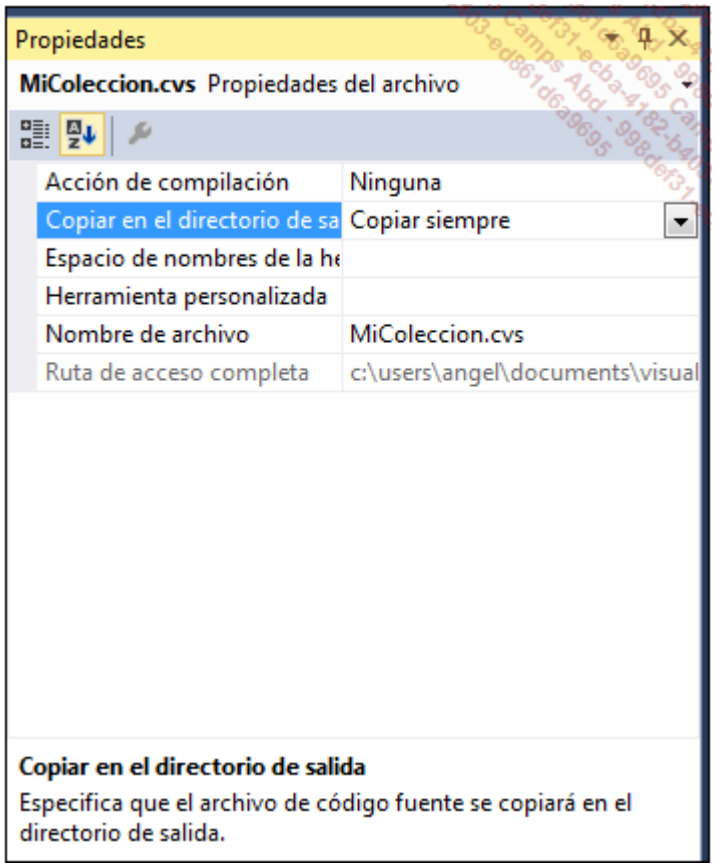


Algunos registros se escriben en el archivo CSV para la demostración, pero podríamos imaginar un contenido generado mediante la exportación de una tabla Excel.

```
ID,Nombre,Apellido,FechaNacimiento,CodigoPostal
1,Ángel,Martín,10/02/59,76300
2,Oscar,Alonso,25/07/1980,76520
3,Martín,Mateo,12/06/200,75001
```

La primera línea del archivo CSV da nombre a los campos de los registros.

Cada ejecución de prueba es objeto de un directorio que contiene el "paquete" comprobado y los resultados. Hay que configurar el proyecto para que el archivo CSV se copie en el directorio de salida del paquete a probar.



Por último, hay que completar el atributo añadido al método de prueba para definir el enlace con el archivo CSV. Esta definición varía en función del tipo básico de los datos. Para un archivo CSV, la definición es la siguiente:

```
[TestMethod,

    DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",

        "|DataDirectory|\\MiColeccion.csv",

        "MiColeccion#csv",

        DataAccessMethod.Sequential)]

public void MiPruebaConCsv(){...}
```

El acceso a los campos del registro se hace con la propiedad DataRow del objeto PruebaContexto. Pero todavía hay que tener una referencia al objeto PruebaContexto.

Cuando una clase de pruebas expone una propiedad PruebaContexto de tipo TestContext, esta propiedad se actualiza antes de la ejecución de una prueba:

```
public TestContext PruebaContexto { get; set; }
```

Con esta referencia actualizada es posible acceder a los campos tanto por su índice como por su nombre, definido en la primera línea del archivo CSV.

```
[TestMethod,
    DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
        "|DataDirectory|\\MiColeccion.csv",
        "MiColeccion#csv",
        DataAccessMethod.Sequential)]
public void MiPruebaConCsv()
{
    if( !string.IsNullOrEmpty(PruebaContexto.DataRow[0].ToString()))
        Trace.WriteLine("MiPruebaConCsv con "
            + string.Format("({0}) Nombre: {1}, Apellido: {2},
            FechaNacimiento: {3}, CodigoPostal: {4}",
            PruebaContexto.DataRow[0],
            PruebaContexto.DataRow["Nombre"],
            PruebaContexto.DataRow["Apellido"],
            PruebaContexto.DataRow["FechaNacimiento"],
            PruebaContexto.DataRow["CodigoPostal"]
            ));
}

public TestContext PruebaContexto { get; set; }
```

La ejecución de esta prueba en modo **Debug** muestra la siguiente traza:

```
MiPruebaConCsv con (1) Nombre: Ángel, Apellido: Martín, FechaNacimiento:
10/02/1959 00:00:00, CodigoPostal: 76300
MiPruebaConCsv con (2) Nombre: Oscar, Apellido: Alonso, FechaNacimiento:
25/07/1980 00:00:00, CodigoPostal: 76520
MiPruebaConCsv con (3) Nombre: Martín, Apellido: Mateo, FechaNacimiento:
12/06/0200 00:00:00, CodigoPostal: 75001
```

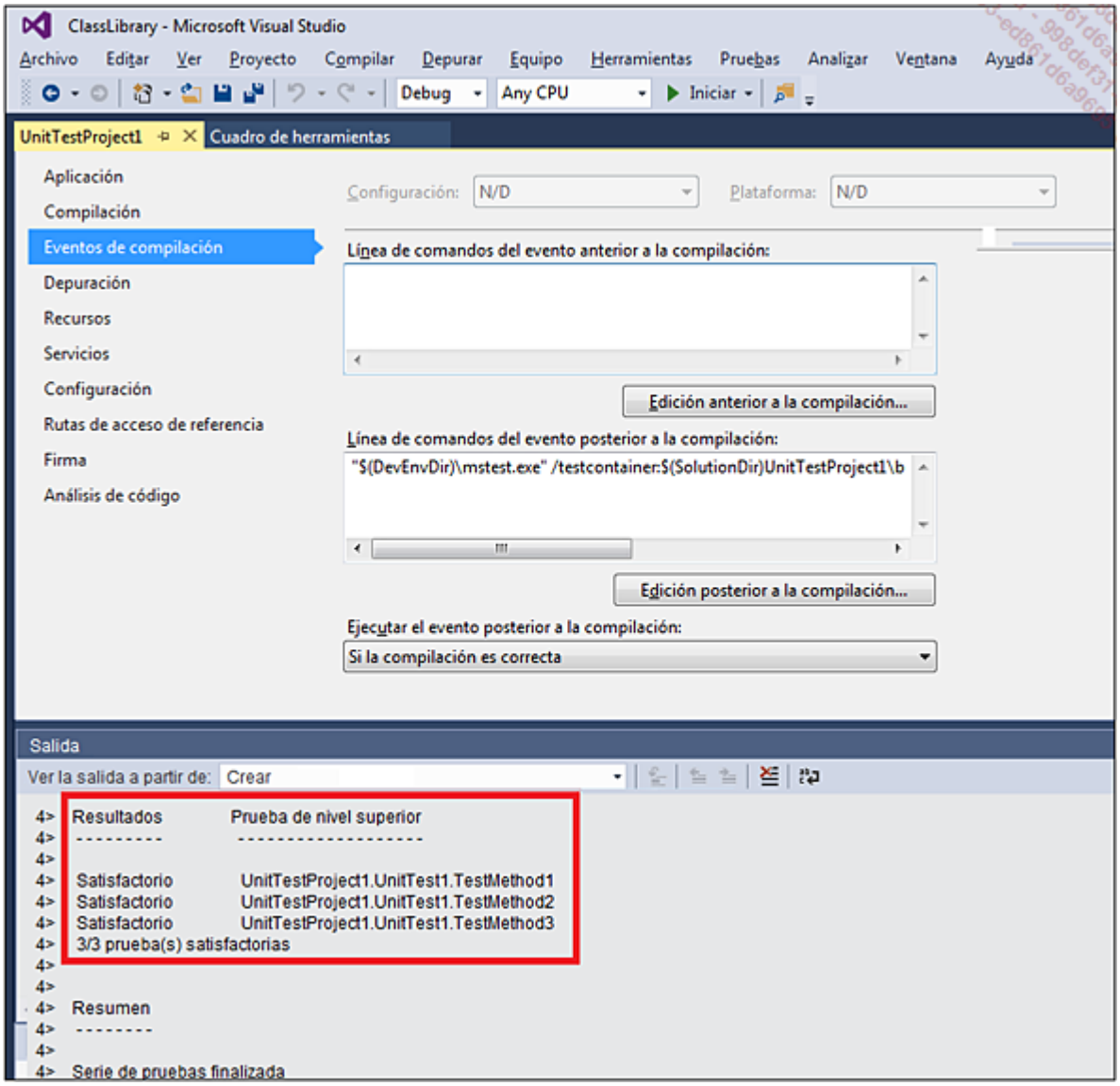
La prueba se llama tantas veces como registros reales haya en la tabla (la primera línea se considera como una línea de definición). El desarrollador puede concentrarse en la redacción de la prueba y en su validación, seleccionando una colección de argumentos bien escogidos.

Los fragmentos de código anteriores forman parte de la solución ClassLibrary disponible para su descarga.

Automatización de las pruebas en la compilación

La calidad de un módulo de software empieza con una compilación sin errores ni advertencias (warning), y continúa con la superación de las pruebas de regresión.

Para automatizar esto en el puesto de desarrollo se pueden utilizar las propiedades **Eventos de compilación** del proyecto de pruebas.



La línea de comando del evento post-build es:

```
"$(DevEnvDir)\mstest.exe"
    /testcontainer:$(SolutionDir)TestClassProject1\bin\$
(Configuration)\$(TargetFileName)
    /noresults
    /detail:errormessasge
```

Utiliza la herramienta Microsoft mstest.exe.

Los fragmentos de código anteriores forman parte de la solución ClassLibrary disponible para su descarga.

Automatización de las pruebas fuera de Visual Studio

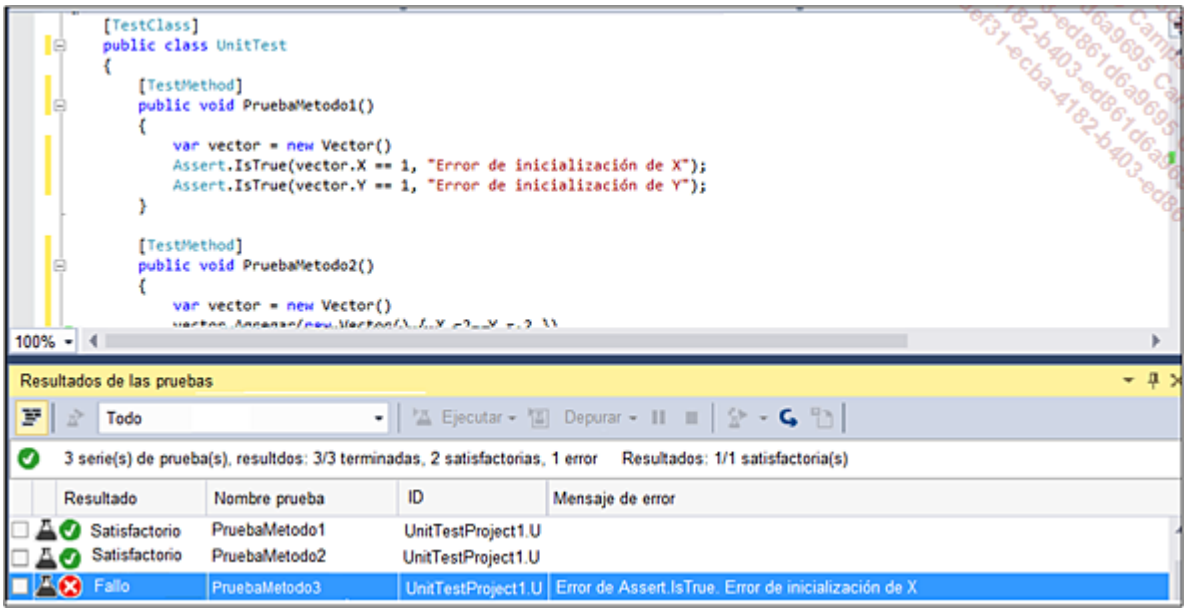
Microsoft ofrece un software, **Tests Agent**, que permite ejecutar las pruebas en una máquina sin tener Visual Studio instalado. La ejecución se realiza desde una consola. Por supuesto, los ensamblados a comprobar se deben compilar, y también los ensamblados de pruebas.

Se prepara un archivo de consola (.BAT) que contiene las instrucciones de las pruebas a ejecutar.

```
: Definiciones de variables DOS
SET ProgFiles86Root=%ProgramFiles(x86)%
IF NOT "%ProgFiles86Root%"=="" GOTO amd64
SET ProgFiles86Root=%ProgramFiles%
:amd64

set MSDIR=%ProgFiles86Root%\Microsoft Visual Studio
14.0\Common7\IDE\CommonExtensions\Microsoft\TestWindow

set TESTFOLDER=C:\Usuarios\Angel\Documentos\MisPruebas

::::::::::::::::::::::::::::::::::::::::
: Bucle de ejecución de las pruebas seleccionadas

for %%s in (
PruebaMetodo1
PruebaMetodo3
) do (
    "%MSDIR%\vstest.console.exe" %TESTFOLDER%\TestClassProject1.dll /Tests:%%
s  /Logger:trx
)

pausa
```

La ejecución de las pruebas genera archivos de resultados (TRX) que se escriben en el directorio **TestResults**. Estos archivos se pueden copiar en el puesto de desarrollo y abrirse en Visual Studio, que muestra su contenido en su ventana **Resultados de las pruebas**.

En la siguiente pantalla se detecta un error durante una prueba. El desarrollador que cargó el proyecto y abrió los archivos TRX puede llegar al código que provoca el error de manera sencilla haciendo doble clic en la línea **Fallo**.



Los fragmentos de código anteriores forman parte de la solución ClassLibrary disponible para su descarga.

CodedUI

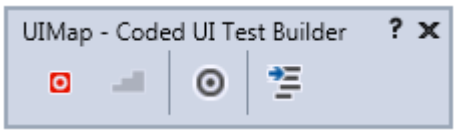
Visual Studio 2015 Entreprise integra **CodedUI**, una herramienta de pruebas con muy buen rendimiento que permite manipular las interfaces de usuario. El desarrollador puede registrar los casos de uso de las especificaciones y reproducirlos como una serie de pruebas gestionadas de manera análoga a las pruebas unitarias.

CodedUI genera objetos que encapsulan los componentes gráficos de la aplicación a probar. Las clases del CodedUI son diferentes de las clases gráficas utilizadas por la aplicación.

El procedimiento de inicio es muy sencillo:

- Agregar a la solución un proyecto de tipo CodedUI.
- Situarse en un método [TestMethod].
- Iniciar el registro desde el menú contextual.

Visual Studio se minimiza y se abre un asistente de aprendizaje, en la parte inferior izquierda de la pantalla.



Pulse el botón **Registrar**. Todas las operaciones del ratón/teclado que se produzcan a continuación se registrarán hasta que pulse el botón **Fin de registro**.

CodedUI habrá creado tantos objetos como sea necesario para reproducir el escenario. El desarrollador se podrá apropiar de este código y completarlo para reforzarlo con Assert, para comprobar que una acción concreta ha provocado la visualización de una información determinada, etc.

Ejercicio

1. Enunciado

Debe escribir

- Una clase ClaseCadena que contenga un método DevuelveIniciales, que permita devolver las iniciales del nombre y apellido que se pasen como argumento en forma de cadena como se muestra a continuación:

```
string iniciales = ClaseCadena.DevuelveIniciales("Ángel Sánchez");

// iniciales debe contenir "Á.S."
```

Si el método recibe un argumento incorrecto, debe devolver una cadena vacía.

- Una serie de pruebas unitarias que permitan comprobar que todos los casos de uso del método no provocan funcionamientos incorrectos.

2. Corrección

La corrección de este ejercicio está en la solución EjercicioDePrueba, disponible para su descarga.

Los casos de error son los siguientes:

- El argumento de tipo string es, por naturaleza, nullable. El valor null, que se pasa como argumento, no debe provocar funcionamientos incorrectos.
- Se debe tener en cuenta el caso de una cadena vacía.
- También se debe comprobar el caso de una cadena que solo contenga una palabra.

```
using Microsoft.VisualStudio.TestTools.TestClassing;
using StringClassLibrary;

namespace TestClassProject
{
    [TestClass]
    public class TestClass1
    {
        /// <summary>
        /// Uso nominal
        /// </summary>
        [TestMethod]
        public void PruebaCasoNormal()
        {
            string iniciales =
ClaseCadena.DevuelveIniciales("Ángel Sánchez");
            // iniciales debe contener "Á.S."
            Assert.IsTrue(iniciales == "Á.S.", "Esperado: Á.S.
Recibido: "+iniciales);
        }

        /// <summary>
        /// Caso de valor null en la cadena
        /// </summary>
        [TestMethod]
```

```
        public void PruebaArgumentoNull()
        {
            string iniciales = ClaseCadena.DevuelveIniciales(null);
            Assert.IsTrue(iniciales == string.Empty,
"PruebaArgumentoNull: Cadena vacía esperada pero no recibida ");
        }

        /// <summary>
        /// Caso de una cadena vacía pasada como argumento
        /// </summary>
        [TestMethod]
        public void PruebaCadenaVacia()
        {
            string iniciales = ClaseCadena.DevuelveIniciales(string.Empty);
            Assert.IsTrue(iniciales == string.Empty,
"PruebaCadenaVacia: Cadena vacía esperada pero no recibida ");
        }

        /// <summary>
        /// Caso de la cadena que solo contiene una palabra
        /// </summary>
        [TestMethod]
        public void PruebaCadenaParcial()
        {
            string iniciales = ClaseCadena.DevuelveIniciales("Andrea");
            Assert.IsTrue(iniciales == string.Empty,
"PruebaCadenaParcial: Cadena vacía esperada pero no recibida");
        }
    }
}
```