MANUALES

USERS .code

C#

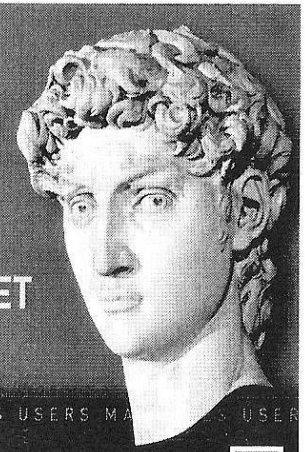
LA GUÍA TOTAL DEL PROGRAMADOR

Desarrollo de aplicaciones bajo el framework .NET

Programación de Web Services

Introducción a Managed DirectX

Acceso a datos externos con ADO.NET



DIEGO RUIZ

ON WEB

LA EVOLUCIÓN DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

Diego G. Ruiz

Es ingeniero en Sistemas de Información, egresado de la Universidad Tecnológica Nacional Argentina. Su interés por la programación comenzó a los 10 años de edad, cuando tuvo en sus manos su primera computadora, la Commodore 64. A partir de aquel momento, inició una carrera que lo llevaría de computadora en computadora y de un lenguaje de programación a otro.
Actualmente se desempeña como programador líder en el departamento de sistemas de la empresa Bioscience, dedicado a la construcción de software para la adquisición, visualización y análisis de señales biológicas. Dicta clases de programación de videojuegos en el instituto Image Campus y es colaborador de la revista de programación users.code en el área multimedia. Es, además, autor del libro *C++ Programación orientada a objetos*.

Dedicatoria

A mi compañera de vida, Ana, quien a pesar de mis defectos, permanece a mi lado.

Agradecimientos

A mi familia, a la que agradezco profundamente su contención y afecto.

SOBRE LA EDITORIAL

MP Ediciones S.A. es una editorial especializada en temas de tecnología (computación, IT, telecomunicaciones).
Entre nuestros productos encontrará: revistas, libros, fascículos, sitios en Internet y eventos.
Nuestras principales marcas son: USERS, Aprendiendo PC, Dr. Max y TecTimes.

Si desea más información, puede contactarnos de las siguientes maneras:
Sitio web: www.tectimes.com
E-mail: libros@mpediciones.com
Correo: Moreno 2062 (C1094ABF), Ciudad de Buenos Aires, Argentina.
Tel.: 54-11-4959-5000 / Fax: 54-11-4954-1791

PRÓLOGO

Desde que Microsoft presentó al mundo su nueva plataforma estratégica de desarrollo, .NET, ésta no ha parado de crecer y extenderse. Hoy en día, forma parte de una de las tecnologías más importantes que un programador debería conocer.

En este libro aprenderá a programar en C#, el lenguaje más importante de la plataforma .NET, el cual le permitirá explotar al máximo sus capacidades.

El lenguaje fue diseñado para crear sistemas en el mundo actual, donde la necesidad de comunicarse con otras aplicaciones y servicios por una red de información –como Internet– es moneda corriente, donde conviven bases de datos de distintos fabricantes, donde los sistemas son construidos a partir de componentes locales o remotos, donde se requieren soluciones integrales para diversos tipos de plataformas.

Con C# podremos crear sistemas de una gran diversidad, simples o complejos, aplicaciones de consola, de escritorio o para la Web, para computadoras personales o para dispositivos móviles, etc. También ofrece una excelente puerta de entrada a quienes deseen iniciarse en la programación orientada a objetos, ya que es fuerte en los aspectos más importantes del paradigma y simplifica las tareas menos agradables que suelen convivir con otros lenguajes más antiguos.

Este libro ofrece una introducción en los aspectos más importantes del lenguaje C# y la plataforma .NET. Pretende ser ideal para el principiante, para el no tan principiante y para los programadores que provengan de otros lenguajes, atraídos por esta grande y maravillosa plataforma de desarrollo. Espero que el lector disfrute tanto su lectura como yo disfruté en escribirlo.

Ing. Diego G. Ruiz
ruizdiego@gmail.com

EL LIBRO DE UN VISTAZO

En la primera parte de este manual, desarrollaremos los conceptos fundamentales y las características particulares que hacen a este lenguaje, para luego introducirnos de lleno en la programación de aplicaciones de escritorio y para la Web.

Capítulo 1

EL LENGUAJE C# Y LA PLATAFORMA .NET

En este primer capítulo estudiaremos qué nos ofrece este nuevo lenguaje y por qué habría de interesarnos.

Capítulo 2

FUNDAMENTOS DEL LENGUAJE C#

Conoceremos los conceptos fundamentales de este lenguaje, que serán útiles tanto para quienes den sus primeros pasos en la programación como para quienes vengan de lenguajes como Visual Basic, C++ o Java.

Capítulo 3

CLASES Y OBJETOS

En este capítulo estudiaremos la anatomía de la estructura de datos más importante del lenguaje: la clase.

Capítulo 4

ENCAPSULAMIENTO, HERENCIA Y POLIMORFISMO

Profundizaremos en los conceptos de la programación orientada a objetos, para luego conocer las particularidades de C#, como los métodos virtuales y las clases abstractas.

Capítulo 5

COLECCIONES

El array es uno de los recursos más poderosos que ofrece C#. Analizaremos aquí diferentes tipos de arrays, de una dimensión y multidimensionales: cómo usarlos y los beneficios que ofrecen.

Capítulo 6

DELEGADOS Y EVENTOS

Los delegados y los eventos son elementos muy utilizados en la programación Windows. C# ofrece un mecanismo simple que cumple con este objetivo y que nos servirá para muchísimos otros fines.

Capítulo 7

APLICACIONES CON WINDOWS.FORMS

Finalmente, en este capítulo nos sumergiremos en la creación de aplicaciones de escritorio. Presentaremos los controles básicos de la librería, y cuáles son sus métodos y propiedades, que nos permitirán crear, desde cero, una aplicación con interfaz de usuario.

Capítulo 8

ACCESO A DATOS EXTERNOS

Veremos cómo acceder y emplear la información almacenada en archivos de texto y binarios, y también en bases de datos relacionales por medio de ADO.NET.

Capítulo 9

MANEJO DE EXCEPCIONES

C# ofrece un sistema de manejo de excepciones muy completo, que nos permitirá lidiar con estas situaciones imprevistas de un modo muy sencillo y efectivo. En este capítulo descubriremos cómo sacar el máximo provecho de él.

Capítulo 10

CARACTERÍSTICAS AVANZADAS DEL LENGUAJE

Conoceremos los aspectos avanzados con los que cuenta el lenguaje: boxing/unboxing, qué son y para qué se utilizan los atributos, qué es Reflection y cómo acceder a librerías creadas en otros lenguajes, entre otros temas.

Capítulo 11

SERVICIOS WEB

Sin dudas, los servicios web son uno de los aspectos de la plataforma que más han crecido en popularidad en los últimos tiempos. Veremos de qué trata esta tecnología y cómo podremos utilizarla desde C#.

Capítulo 12

SOCKETS

En este capítulo, desarrollaremos la programación de sockets, y podremos hacer que nuestra aplicación se comunique con otra por medio de una red de datos.

Capítulo 13

INTRODUCCIÓN A MANAGED DIRECTX

Realizaremos una breve introducción a Managed DirectX, una suma de clases para .NET que nos permitirán hacer uso de DirectX.

Apéndice A

DOCUMENTACIÓN DE CÓDIGO

Conoceremos qué herramientas nos ofrece C# para llevar adelante esta ardua pero importante tarea.

Apéndice B

LA NOTACIÓN HÚNGARA

En este apartado veremos los conceptos fundamentales de la notación Húngara. Básicamente, se trata de una convención de nombramiento de variables y funciones muy utilizada en C++, que tiene una menor incidencia en lenguajes como C# y Java.

Apéndice C

UML

Descubriremos los fundamentos del lenguaje de modelado unificado, un sistema notacional destinado al modelado de aplicaciones que se basen en conceptos de la programación orientada a objetos.

! INFORMACIÓN COMPLEMENTARIA

A lo largo de este manual encontrará una serie de recuadros que le brindarán información complementaria: curiosidades, trucos, ideas y consejos sobre los temas tratados.
Cada recuadro está identificado con uno de los siguientes iconos:

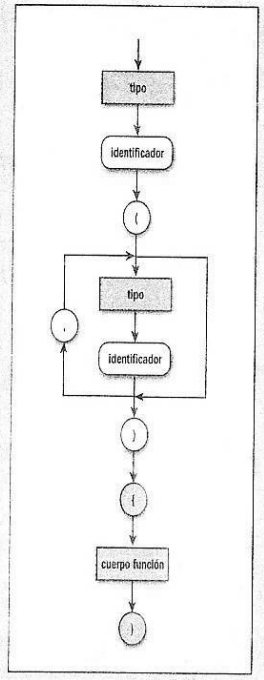

CURIOSIDADES E IDEAS


ATENCIÓN


DATOS ÚTILES Y NOVEDADES

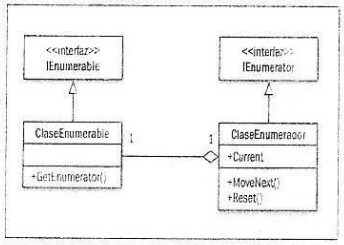

SITIOS WEB

CONTENIDO

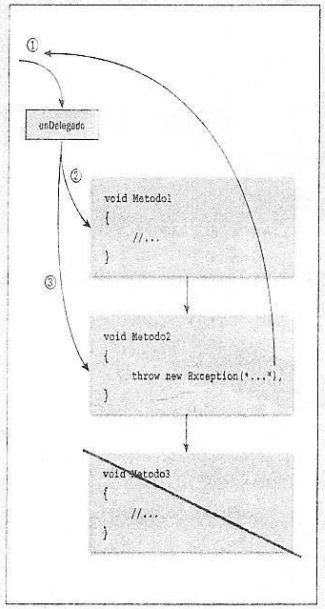


Capítulo 4

Capítulo 5

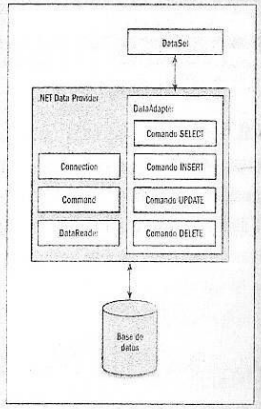


Capítulo 6



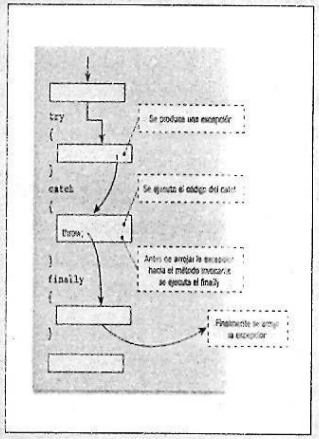
Capítulo 7

Capítulo 8



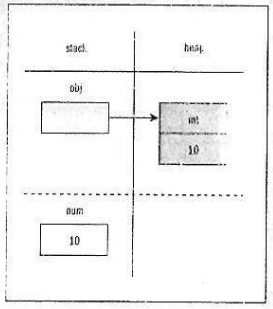
Capítulo 9

MANEJO DE EXCEPCIONES



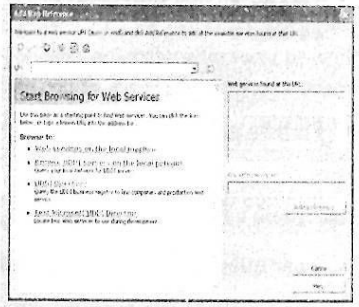
Capítulo 10

CARACTERÍSTICAS AVANZADAS DEL LENGUAJE



Capítulo 11

SERVICIOS WEB

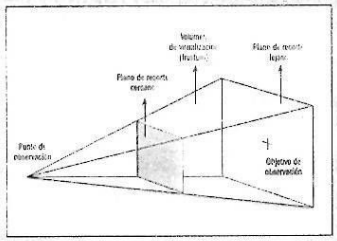


Capítulo 12

SOCKETS

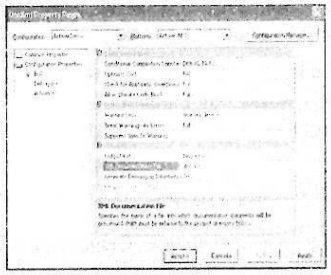
Capítulo 13

INTRODUCCIÓN A MANAGED DIRECTX



Apéndice A

DOCUMENTACIÓN DE CÓDIGO

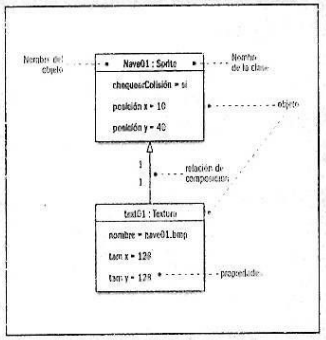


Apéndice B

LA NOTACIÓN HÚNGARA

Apéndice C

UML



Servicios al lector

INTRODUCCIÓN

Este libro se encuentra dirigido, principalmente, al programador novato y de nivel medio, así como al estudiante o al aficionado autodidacto.
Quienes estén dando sus primeros pasos en el mundo de la programación no podrán dejar de sumergirse en los primeros dos capítulos, donde se realiza una introducción a la plataforma .NET y a la sintaxis básica del lenguaje. También ambos resultarán muy útiles para quienes provengan de otros lenguajes de programación, ya que se evidencian las diferencias sustanciales que éstos poseen respecto a C#.

Si acaso a usted le atrae la programación orientada a objetos y desea conocer sus conceptos fundamentales y cómo C# la soporta, los capítulos 3 y 4 le serán de gran utilidad.
Si, en cambio, desea crear aplicaciones de escritorio, no puede dejar de visitar el capítulo 7, donde veremos cómo generar este tipo de programas haciendo uso de la gran diversidad de controles que la plataforma .NET nos ofrece. Pero claro, ninguna aplicación es demasiado útil si no puede interactuar con datos externos. En el capítulo 8 veremos de qué trata ADO.NET y cómo nos facilita la tarea de trabajar con diversas fuentes de datos.

¡Y esto no termina aquí! C# y la plataforma .NET son tan versátiles, que nos permitirán publicar los servicios de una aplicación en una red de un modo muy sencillo. En el capítulo 11 veremos cómo generar servicios web, y en el capítulo 12 aprenderemos a crear aplicaciones de red utilizando sockets.
¿Desea crear aplicaciones gráficas de alto desempeño? ¿Acaso le interesa programar juegos? Entonces la librería Managed DirectX será una gran aliada en su misión, y el capítulo 13 ofrece una introducción a sus principales características.

Y como si todo esto fuera poco, en los apéndices encontrará cómo crear documentación a partir del código escrito en C#, de qué trata la notación de nombramiento Húngara, un listado de lenguajes .NET que podrá encontrar en la Red –¡son más de cincuenta!– y, por último, una introducción al lenguaje unificado de modelado (UML).
No es exagerado enunciar que C# y la plataforma .NET son el sueño de todo programador. Si acaso duda de mi palabra, ¿qué espera para comprobarlo?

Ing. Diego G. Ruiz
ruizdiego@gmail.com

Capítulo 1

El lenguaje C# y la plataforma .NET

En este primer capítulo estudiaremos qué nos ofrece este nuevo lenguaje y por qué habría de interesarnos. Realizaremos una comparación con lenguajes similares y analizaremos cuál es la arquitectura de la plataforma sobre la que está construido.

¿OTRO LENGUAJE NUEVO?

C# irrumpe en el mercado como un lenguaje muy bien diseñado y con muchas virtudes en una industria plagada de soluciones y herramientas de programación para todos los gustos. ¿Cuáles son, entonces, los motivos por los cuales deberíamos optar por C#?

- C# es un lenguaje moderno y orientado a objetos, con una sintaxis muy similar a la de **C++** y **Java**. Combina la alta productividad de **Visual Basic** con el poder y la flexibilidad de C++.
- La misma aplicación que se ejecuta bajo **Windows** podría funcionar en un dispositivo móvil de tipo **PDA**. Con C#/.NET no nos atamos a ninguna plataforma en particular.
- Se puede crear una gran variedad de aplicaciones en C#: aplicaciones de consola, aplicaciones para Windows con ventanas y controles, aplicaciones para la Web, etc.
- C# gestiona automáticamente la memoria, y de este modo evita los problemas de programación tan típicos en lenguajes como C o C++.
- Mediante la plataforma .NET desde la cual se ejecuta es posible interactuar con otros componentes realizados en otros lenguajes .NET de manera muy sencilla.
- También es posible interactuar con componentes no gestionados fuera de la plataforma .NET. Por ello, puede ser integrado con facilidad en sistemas ya creados.
- Desde C# podremos acceder a una librería de clases muy completa y muy bien diseñada, que nos permitirá disminuir en gran medida los tiempos de desarrollo.

Pero ¿dónde quedan los demás lenguajes? ¿Qué motivó a Microsoft a desarrollar la plataforma .NET?

Durante algún tiempo, cuando la programación en plataforma PC/Windows se popularizó, los caminos más comunes eran, principalmente:

- **Visual Basic**: un lenguaje fácil de aprender pero con muchos defectos. Gran parte de esas deficiencias es fruto de su afanoso objetivo por ser sencillo para el programador novato. Es un lenguaje orientado a objetos *light*. Posee algunas de las características más populares de la POO implementadas, pero muchas de ellas (las que realmente extrañaremos en proyectos complejos) permanecen ausentes, como la herencia, los métodos virtuales, la sobrecarga de operadores, etc.
  Claro que VB también posee muchas virtudes. Realizar una aplicación Windows nunca había sido tan fácil, y si ciertas tareas se encuentran fuera del alcance del lenguaje, es posible realizar un componente en, por ejemplo, C++ y utilizarlo desde VB sin inconvenientes.
- **Visual C++**: es ideal para crear componentes, librerías y drivers, pero la productividad desciende abruptamente cuando se trata de aplicaciones con formularios complejos. Es que, para esto, **Visual C++** se basa en MFC (*Microsoft Foundation Classes*, un grupo de clases que encapsulan el API de Win32 y agregan algunas funcionalidades para facilitar la creación de aplicaciones). Diseñar formularios con MFC es una tarea poco grata; muchas de las propiedades de los controles de esta librería deberán ser fijadas en tiempo de ejecución, lo que aumentará la cantidad de código que deberá poseer nuestra aplicación.
- **Delphi** y **C++ Builder**: basados en el lenguaje de programación **Pascal** y C++, respectivamente, y ambos haciendo uso de una librería llamada **VCL** (*Visual Component Library*), permiten la creación de formularios complejos de una manera bastante sencilla, y poseen la capacidad de fijar casi la totalidad de las propiedades necesarias en tiempo de diseño y con base en dos de los lenguajes más exitosos de todos los tiempos.

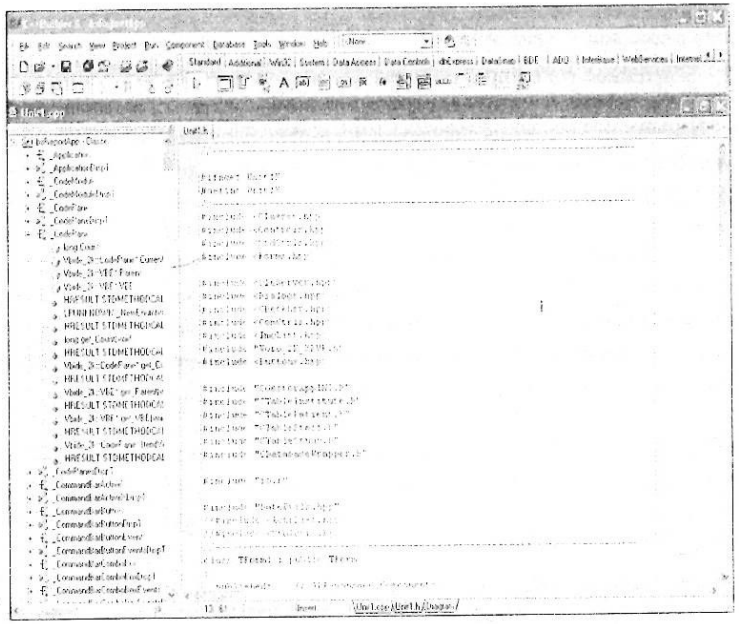


***Figura 1.** El entorno de programación **Borland C++ Builder 6.0**.*

ONWEB.TECTIMES.COM

En el sitio **onweb.tectimes.com** encontrará contenido adicional vinculado a los temas tratados en este manual, incluyendo el código fuente de los ejemplos que se desarrollarán en este libro, listo bajar y probar, así como una completa guía de sitios web recomendados.

Pero no todo es color de rosa para estos entornos. Las aplicaciones son cada día más complejas y los programadores requieren, cada vez en mayor medida, un buen soporte a la colaboración entre componentes. Sería deseable poder emplear desde **Delphi** una librería C++ ya existente, de una manera sencilla, sin tener que estar utilizando herramientas de conversión.

En el caso de C++ Builder, hasta existen inconvenientes para acceder a librerías creadas en Visual C++ debido a que utilizan formatos de librerías distintos (**OMF** en el caso de C++ Builder y **COFF** en el caso de Visual C++), y esto es malo dado que Visual C++ es el entorno más popular (por lejos) para la creación de librerías para Windows, y muchísimos recursos que podemos encontrar en la Red fueron creados con este entorno.

Si deseamos, por ejemplo, crear una aplicación que utilice DirectX desde C++ Builder, tendremos que buscar en la Red las librerías **.LIB** correspondientes en formato OMF (existe una herramienta provista por el entorno para la conversión de formato COFF a OMF, pero la importación de librerías complejas, como lo es DirectX, no es para nada trivial).

Finalmente, en los últimos años ha surgido una gran variedad de plataformas móviles muy poderosas: PDAs, teléfonos celulares, Tablet PCs, etc. Estos dispositivos no son compatibles con binarios creados para microprocesadores Intel x86, ya que poseen su propia familia de microprocesadores con su propio paquete de herramientas para la construcción de aplicaciones nativas.

Ni Delphi ni C++ Builder se encuentran presentes en otras plataformas más allá de Windows (y Linux, si tomamos en consideración a Kylix, que es un entorno de desarrollo creado por Borland, muy similar a Delphi y a C++Builder). Visual Basic y Visual C++ se hallan en versiones especiales para algunos dispositivos móviles, pero el subconjunto de funciones del API de Win32 que están en dichos dispositivos hace que la migración de una aplicación de una plataforma se convierta en una tarea bastante engorrosa.

BORLAND KYLIX

Kylix es un entorno de programación para sistemas Linux muy similar a Delphi y a C++ Builder. Con él es posible crear aplicaciones con ventanas y controles (botones, cuadros de texto, cuadros de lista, etc.) y compilar en Windows y en Linux. Más info: **www.borland.com/kylix**.

.NET

.NET es un conjunto de tecnologías construidas a partir de una estrategia de Microsoft para la cual ha destinado gran parte de su presupuesto de investigación. Esta estrategia surge, a su vez, del nuevo mapa de necesidades y requerimientos que Microsoft advirtió que se suscitarían en el futuro.

.NET son aplicaciones de servidor (SQL Server 2000, Exchange 2000, etc.), es un entorno de desarrollo (Visual Studio .NET), son componentes clave que se integran al sistema operativo, son servicios, y es, finalmente, una plataforma de desarrollo denominada **framework .NET**.

El framework .NET

Es una infraestructura de desarrollo que está compuesta por diversos recursos, entre los cuales se destaca el más importante, que es una máquina virtual conocida como **CLR** (*Common Language Runtime*), sobre la cual se ejecutan las aplicaciones. De este modo, nuestros programas ya no poseerán código nativo de ningún microprocesador en particular, sino instrucciones **MSIL** (*Microsoft Intermediate Language*) que serán traducidas a código nativo en el momento de su ejecución (por medio de un compilador *Just In Time*).

El framework también define una librería base de clases, **BCL** (*Base Class Library*), a la cual puede acceder cualquier desde lenguaje desarrollado para la plataforma.

Por encima de la infraestructura se ubicará un conjunto de reglas básicas que debe implementar un lenguaje para poder ser parte de la familia .NET. Esta especificación es conocida como **CLS** (*Common Language Specificeation*).

Finalmente, se encuentra el conjunto de lenguajes que cumplan con la especificación CLS, como el C#, el VB.NET, Managed C++, etc.

.NET FRAMEWORK SDK

El desarrollo del framework .NET no se detiene. Actualmente la versión 2.0 se encuentra en etapa beta. Es posible acceder a más información acerca de esta nueva versión en el sitio oficial: **http://msdn.microsoft.com/netframework**.

Es posible crear recursos en cualquiera de estos lenguajes que hagan uso de recursos escritos en otros; de hecho, es posible crear una clase en C# que herede de una clase creada en Managed C++.

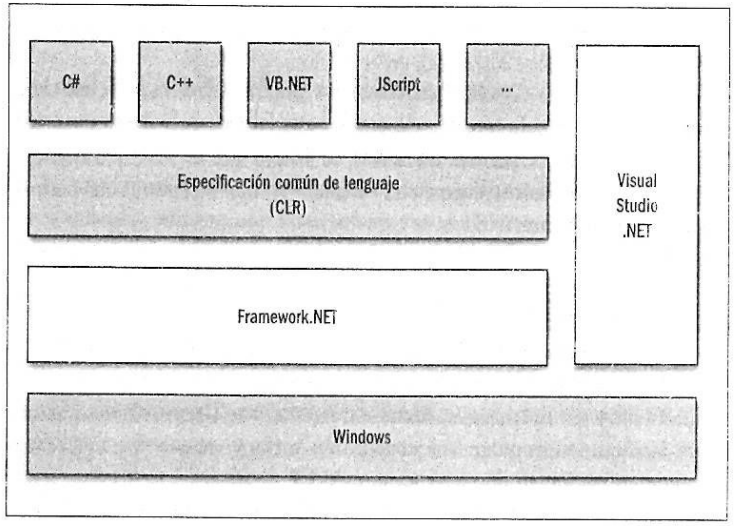


Figura 2. *Arquitectura de la infraestructura de desarrollo.*

EL LENGUAJE C#

C# es un lenguaje que cumple con la especificación CLS. El código que crearemos con él será traducido a instrucciones **MSIL** para entonces ser traducido, justo antes de su ejecución, a instrucciones nativas que correspondan a la plataforma concreta sobre la cual estemos trabajando.

Cabe destacar que el compilador JIT (*Just In Time*) traduce el código MSIL a código nativo no de manera monolítica, sino por métodos, módulos y componentes. Por lo tanto, a grandes rasgos: código que no sea ejecutado no será compilado.

El código MSIL generado a partir de la compilación de código C# es idéntico al código MSIL generado a partir de cualquier otro lenguaje CLS. Esto podría abrir el interrogante de ¿por qué programar en C# en lugar de hacerlo en VB.NET o en Managed C++ o, incluso, en Delphi .NET? Esta pregunta podría responderse con otra: ¿por qué programar en C++ en lugar de hacerlo en C o Pascal, o en cualquier otro lenguaje compilado, si todos generan el mismo código Intel x86?

Cada lenguaje posee sus características que lo tornan ideal para ciertos usos; además, presenta diversos grados de expresividad que pueden permitir implementar el mismo algoritmo de maneras diversas, por lo que un modo puede resultar más eficiente que otro.

Características fundamentales del lenguaje

C# es un lenguaje moderno y altamente expresivo que se ajusta al paradigma de programación orientada a objetos. Su sintaxis es similar a C++ y Java. El lenguaje fue desarrollado en gran parte por **Anders Hejlsberg** (creador del mítico compilador **Turbo Pascal**[1] y uno de los diseñadores líder del lenguaje de programación Delphi).

En C# no existe el concepto de función global o variable fuera de una clase u objeto. Por su buen apego a la POO, es posible sobrecargar métodos y operadores. Soporta definición de interfaces. Ninguna clase puede poseer más de un padre (no se permite la herencia múltiple), pero sí puede suscribir un contrato con diversas interfaces. Soporta la definición de estructuras, pero, a diferencia de C++, aquí no son tan parecidas a las clases y poseen ciertas restricciones que veremos luego.
Permite además la declaración de propiedades, eventos y atributos (que son construcciones declarativas).
Por último, una característica distintiva cada vez más presente en lenguajes modernos es la gestión automática de memoria y el uso de referencias en lugar de punteros. Gracias a esta gestión automática de memoria ya no tendremos que preocuparnos por la existencia de *memory leaks* (zonas de memoria que permanecen reservadas pero ya no son utilizadas debido a errores de programación).

A continuación realizaremos una breve comparación de C# con los lenguajes más populares del momento:

C# contra Visual Basic 6.0

Si usted es un programador Visual Basic y está evaluando "moverse" a C#, no lo dude un segundo. Existen muchísimas razones para tomar esta decisión, y aquí expondremos algunas de ellas. En primer lugar, uno de los motivos por los cuales Visual Basic era atractivo era su productividad. Sí, a pesar de algunos de sus inconvenientes para crear aplicaciones sencillas, no existía mejor solución que Visual Basic; ningún

1 No confundir con el autor del lenguaje Pascal, que fue Niklaus Wirth. Anders Hejlsberg fue el creador de un compilador llamado "Turbo Pascal", que fue muy popular en sus tiempos por su velocidad y bajo costo.

otro lenguaje podía competir en velocidad de desarrollo con él. Pues bien, C# le ha quitado la corona; la misma aplicación que usted puede realizar en Visual Basic podrá crearla en C# en, al menos, el mismo tiempo, e incluso más rápidamente. Por otro lado, gracias a las nuevas características del lenguaje, el diseño de sus aplicaciones podrá ser más elegante y simple, y de este modo podrá *manejar* la complejidad de una manera más natural.

En segundo lugar, seamos sinceros: los programadores Visual Basic nunca fueron vistos como programadores reales, aunque muchísimos profesionales sufran de esta etiqueta de manera injusta. Los sueldos de un programador Visual Basic son mucho más bajos que los de un programador C++ o Java. C#, en cambio, es visto como un lenguaje más profesional. Claro que éste es un punto un tanto controvertido, y podríamos discutir las razones del porqué de esta situación, pero, dejando excepciones a un lado, es una realidad de mercado.

Por último, el pasaje de Visual Basic 6.0 a C# hasta podría ser considerado lógico para quien sea un dominador del VB; es como comprarse un automóvil más lujoso o mudarse a una casa más grande: es el paso evolutivo natural.

Probablemente, lo que usted estará preguntándose respecto al pasaje de Visual Basic 6.0 a C# es el natural ¿para qué existe Visual Basic .NET? Bueno, tal vez Visual Basic .NET nunca debió haber existido; es "demasiado" distinto de Visual Basic 6.0 para ser considerado una nueva versión del lenguaje y, a fin de cuentas, sigue siendo Basic. Tal vez la pregunta que podríamos hacernos sería: si es que vamos a tener que aprender un lenguaje nuevo, ¿por qué no aprender el mejor y más popular lenguaje de la plataforma?

C# contra C++

Admito que C++ es mi lenguaje preferido; trabajé durante muchos años con él e, incluso, escribí un libro de C++ (*C++ Programación Orientada a Objetos*, de esta misma editorial), pero probé C# y fue como probar ambrosía.

VISUAL BASIC .NET VS. C#

En el enlace de referencia es posible encontrar una comparativa entre Visual Basic .NET y C# escrita por Mario Félix Guerrero.

Enlace: **www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_2128.asp**.

Hoy día, en mi trabajo diario, complemento de manera natural ambos lenguajes; no fue difícil ingresar en el mundo C# desde C++. Lo que debo admitir es que luego de estar días trabajando con C#, volver a C++ es un cambio un tanto abrupto, y me descubro preguntando una y otra vez por qué no se encontrará disponible tal o cual característica de C# en C++.

Los detractores de C++ lo acusan de ser un lenguaje híbrido, un C con clases, como muchas veces es definido. El problema de esta falta de decisión en cuanto al diseño del lenguaje es que permite que convivan clases con funciones y variables globales, lo cual podría ser tentador para el programador y podría empobrecer el diseño de una aplicación construida de este modo. Claro que, por otro lado, hay quienes gustan de ver el vaso medio lleno y sostienen que esta característica de "hibridez" es positiva, pues deja al programador la posibilidad de tomar lo mejor de los dos mundos.

C# es, en alguna medida, una evolución de C++, ya que toma prestada su sintaxis y mejora muchos de sus aspectos, como el hecho de poseer una librería de clases unificada y ser un lenguaje orientado a objetos puro (adiós hibridez). Esto no quiere decir que lo reemplace en todos sus usos, claro. C++ seguirá teniendo su segmento de utilización en donde es ideal, pero C# invade día tras día áreas donde antes C++ era único conquistador.

C# contra Java

Muchos sostienen que C# es una copia de Java. Si tomamos ciertos trozos de código (convenientemente) escritos en alguno de estos lenguajes, podría ser imposible determinar si es C# o es Java.

Lo cierto es que parte de la filosofía empleada por ambos lenguajes es la misma. Ambos se ejecutan sobre una máquina virtual, y esta característica los convierte en lenguajes potencialmente multiplataforma; ambos poseen especificaciones de subconjuntos de recursos de lenguaje para implementaciones en diversos dispositivos (por ejemplo, móviles), y ambos tienen una librería de clases con muchas características en común.

SIMILITUDES ENTRE C#, C++ Y JAVA

En el enlace de referencia podrán encontrar una breve comparativa entre estos lenguajes. Enlace: **www.microsoft.com/spanish/MSDN/estudiantes/desarrollo/lenguajes/c-sharp.asp**.

Sin embargo, C# presenta la ventaja de integrarse mejor con aplicaciones nativas de la plataforma sobre la cual estemos trabajando. Claro que, si accedemos a recursos nativos, perderá la característica de ser multiplataforma, pero esta característica no siempre es deseada. Podríamos tener la necesidad de invocar métodos de librerías nativas creadas en C++ para Windows de una manera sencilla y eficiente.

Luego podríamos discutir en muchos puntos cuál es mejor que cuál; en la Red, los foros de discusión sobre programación se encuentran plagados de peleas de este tipo. La realidad es que hoy día C# es más fuerte en plataformas basadas en Windows, mientras que Java es más fuerte en una gran diversidad de plataformas menos populares (celulares, tarjetas inteligentes, etc.) y es el candidato ideal si hoy desea construir una aplicación que deba ejecutarse sin cambios en Linux y Windows, aunque esta realidad se encuentre próxima a cambiar.

EL ENTORNO DE DESARROLLO VISUAL STUDIO .NET

En el desarrollo de este libro trabajaremos con Visual Studio .NET 2003, que es la última versión de producción del entorno en el momento de escribir estas líneas. Existe una versión "2005 Express" en estado beta que también podría utilizar el lector.

Por otro lado, el libro se basa principalmente en la versión 1.1 del framework, que también es la última versión de producción a la fecha de publicación de esta obra. Quienes no posean el entorno Microsoft Visual Studio pueden utilizar **Mono** en ambientes **Windows**, **GNU/Linux** y **Mac Os X**.

El entorno Visual Studio .NET 2003 es una excelente herramienta de desarrollo. Con ella podremos crear soluciones que, a su vez, podrán contener uno o más proyectos de diversos lenguajes en función de los paquetes que tengamos instalados; en

VISUAL STUDIO 2005 EDICIÓN EXPRESS

Es posible descargar la versión beta de este entorno desde **http://msdn.microsoft.com/vs2005**. Seguramente, en el transcurso del año, el entorno saldrá en su versión definitiva y ya no será posible descargar la versión beta de manera gratuita. De todos modos, Microsoft planea ofrecer las versiones express del entorno de desarrollo a un costo alcanzable por cualquier desarrollador.

principio dispondremos de C#, Visual Basic .NET y C++. También es posible crear aplicaciones C# utilizando el framework .NET SDK, que es un conjunto de herramientas que nos permiten compilar código fuente C# para crear aplicaciones. Visual Studio .NET hace uso del framework .NET SDK y funciona como *front end* para evitar que nosotros tengamos que interactuar con herramientas en comando de línea. El framework .NET SDK puede ser descargado desde el siguiente enlace: **http://msdn.microsoft.com/netframework**.

Sin embargo, Visual Studio .NET es mucho más que un simple *front end*. Algunas de las tareas que podremos realizar con él son las siguientes:

- Navegar fácilmente por las clases por medio del visor de clases.
- Navegar por los archivos de nuestros proyectos por medio del explorador de soluciones.
- Entender más rápidamente el código escrito gracias a que el editor colorea las palabras reservadas y los tipos de datos conocidos.
- Organizar múltiples proyectos y editar fácilmente sus propiedades.
- Configurar el entorno para que ejecute herramientas externas (como precompiladores).
- Depurar nuestros proyectos fácilmente y consultar valores de objetos de modo interactivo, así como realizar depuraciones remotas desde otras computadoras.
- Acceder a facilidades de búsqueda y reemplazo por hoja de código fuente activo y en archivos.
- Editar recursos (bitmaps, iconos, archivos binarios, etc.) por medio de herramientas integradas, y navegar por ellos por medio del visor de recursos.
- Agregar tareas por realizar haciendo uso de la lista de tareas pendientes, que además inserta automáticamente tareas a partir de comentarios prefijados.
- Generar documentación en formato HTML por medio de una herramienta provista con el entorno. Esta documentación es generada a partir del código fuente y comentarios con formatos específicos que escribiremos en él.
- Colapsar y expandir trozos de código para mejorar la legibilidad de nuestras fuentes.
- Posibilidad de integrar herramientas al entorno por medio de un sistema de plug-ins.

¿QUÉ ES MONO?

Desde el año 2001, la empresa **Ximian** (adquirida luego por **Novell**) comenzó a desarrollar el proyecto **Mono**, que es una plataforma de desarrollo **Open Source** basada en el framework .NET. Con Mono es posible escribir aplicaciones en C# (o VB.NET) y ejecutarlas no sólo en Windows, sino también en **GNU/Linux** y **Mac Os X**. Se puede descargar desde **www.mono-project.com**.

Haciendo uso del entorno de programación Visual Studio .NET crearemos proyectos. Un proyecto contendrá, básicamente, hojas de código fuente en C# (luego veremos que también podrá contener otros recursos).

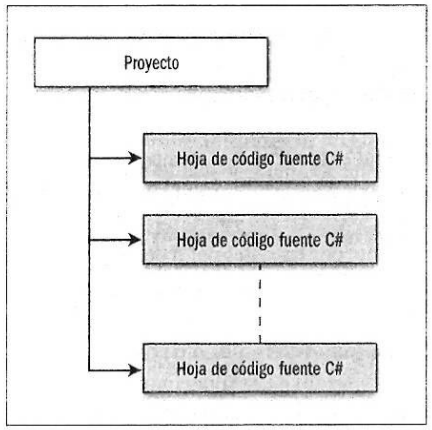


Figura 3. *Organización de un proyecto.*

Un proyecto posee ciertas propiedades, las cuales indican de qué manera deberán compilarse las fuentes y los recursos que incluye. La idea de "proyecto" como organización de fuentes se encuentra muy extendida, y casi todo entorno de programación la maneja. Usualmente, un proyecto construido tendrá como salida una aplicación ejecutable (archivo **.EXE**) o una librería (archivo **.DLL**). Sin embargo, es posible (y, de hecho, muy común) que el sistema en el cual estemos trabajando se encuentre integrado por más de un componente; por ejemplo, una aplicación y una librería de enlace dinámico que sea utilizada por la aplicación. En este caso, no es sólo un proyecto el que deberemos crear, sino dos.

Ahora bien, si necesitáramos crear dos proyectos siendo uno dependiente del otro, ¿qué deberíamos hacer? Podríamos crear un proyecto en una instancia de Visual Studio y otro proyecto en otra instancia, pero ésta no es una muy buena idea, ya

¿QUÉ ES UNA DLL?

Una DLL (*Dynamic Link Library*) es un conjunto de funciones y/o clases que pueden ser accedidas y utilizadas por otros programas en tiempo de ejecución. Estas librerías pueden ser creadas desde C# o desde otros lenguajes.

que si modificamos la librería y nos olvidamos de reconstruirla, nuestra aplicación seguirá utilizando una versión vieja de ésta.

Afortunadamente, Visual Studio .NET soporta más de un proyecto abierto de manera simultánea en el mismo espacio de trabajo. Estos proyectos pueden relacionarse entre sí por medio de dependencias; de modo que si reconstruimos la aplicación, el entorno automáticamente reconstruirá todos los proyectos dependientes que se hayan modificado (por ejemplo, la librería).

Visual Studio denomina **solución** a esta agrupación de proyectos (aunque en ediciones anteriores del entorno se las llamaba **espacios de trabajo**). Todo proyecto debe estar contenido en una solución; por lo tanto, cuando creamos un proyecto nuevo, Visual Studio nos crea automáticamente una solución que lo contiene.

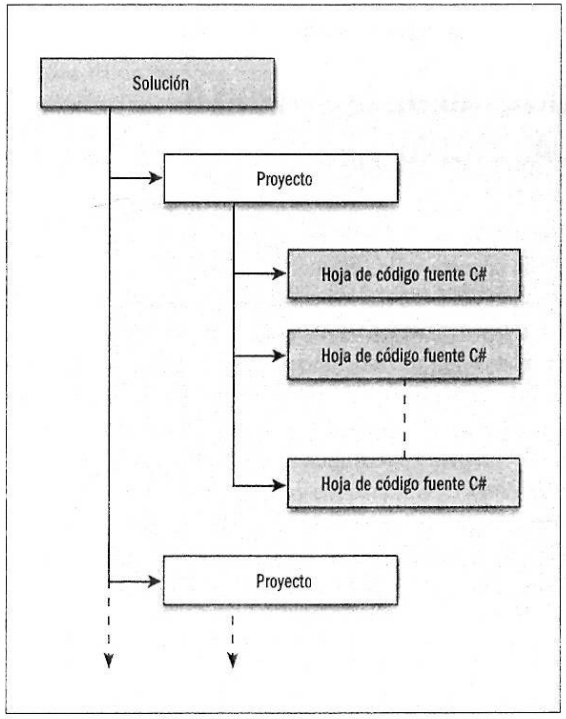


Figura 4. *Organización de una solución.*

Luego, cuando construimos la solución se construyen todos los proyectos (que pueden estar relacionados directamente o no), es decir que se procesan todos los elementos de cada proyecto para generar los archivos de salida que correspondan.

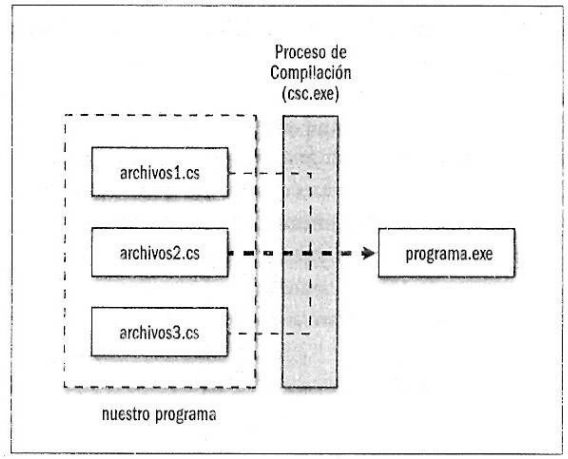


Figura 5. *Compilación de un proyecto.*

Nuestra primera aplicación con Visual Studio .NET

Pero veamos cómo crear nuestra primera aplicación C# con él. En primer lugar, cabe destacar que el entorno es sumamente configurable y que la organización de los paneles, así como la pantalla de inicio, puede variar en función de cómo la configuremos. Si es la primera vez que ejecuta la aplicación o si no ha modificado las opciones de inicio predeterminadas, debería encontrarse con la "página de inicio" que se muestra en la siguiente figura:

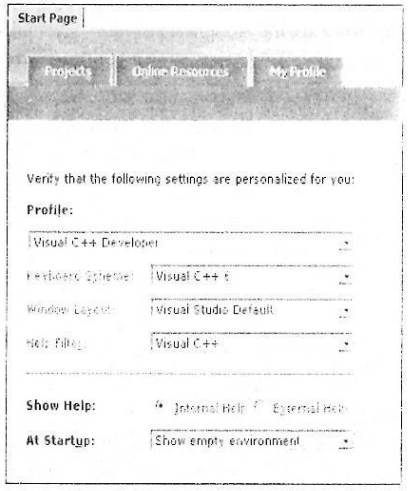


Figura 6. *La página de inicio.*

En lo personal, me desagrada bastante dicha "página", por lo que lo primero que hago es fijar las opciones como indica precisamente la figura mencionada, es decir:

Profile: Visual C++ Developer
At Startup: Show empty enviroment

Cambiar el perfil permite que se modifique la ubicación predeterminada de los paneles en donde se encuentran las herramientas (como el visor de clases, el explorador de soluciones, etc.). En lo personal, me he acostumbrado a tener el panel de visor de clases a la izquierda, como estaba en el viejo y querido Visual C++ 6.0, pero cada quien podrá hacer lo que le plazca, ya que esta configuración no nos impedirá modificar ni crear ningún tipo de proyecto u opción en particular.

Fijar el inicio de aplicación en **Show empty enviroment** (*Mostrar un entorno vacío*) también es una cuestión de gustos, y lo que especifica es que cada vez que iniciemos la aplicación veremos lo que muestra la siguiente figura:

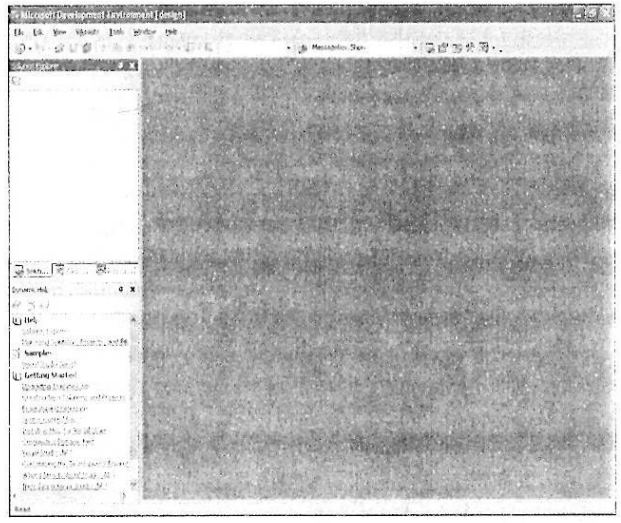
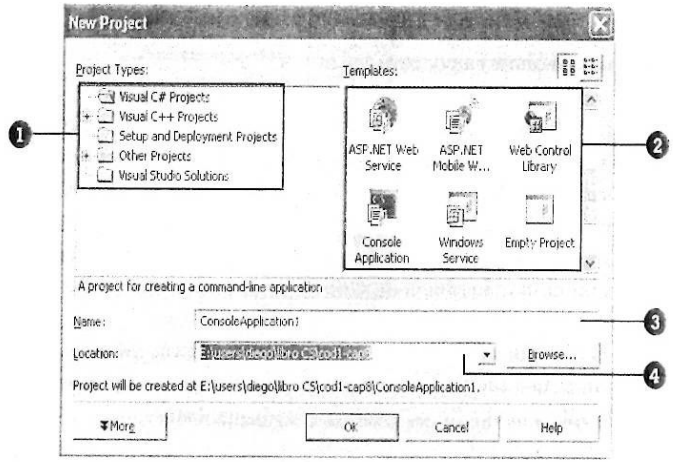


Figura 7. *Visual Studio, comenzando con el entorno vacío.*

Bueno, ahora sí, vayamos a la acción. Teniendo el entorno ya abierto, lo primero que deberemos hacer será "Crear un nuevo proyecto". Si estamos en la "Página de Inicio", deberemos seleccionar la pestaña **Projects**, y luego, presionar el botón **New Proyect** que se encuentra en el extremo inferior de la página. Si estamos en un entorno vacío, deberemos acceder al menú **File**, desde allí seleccionar **New** y, finalmente, la opción **Project**. Hecho esto, deberá aparecer la ventana de la **Guia Visual 1**.

Nuevo proyecto GUÍA VISUAL 1



1. Tipos de proyectos disponibles
2. Lista de plantillas de proyectos
3. Nombre del proyecto
4. Ubicación de la carpeta del proyecto

Seleccionaremos siempre que el tipo de proyecto sea **Visual C# Projects**, luego especificaremos la plantilla del tipo de proyecto que deseemos crear. Lo que hace la plantilla del proyecto es fijar opciones predeterminadas al modo de compilación de nuestro proyecto, que finalmente especificará los parámetros que se enviarán al compilador C# cuando construyamos el proyecto para generar un archivo ejecutable (con instrucciones MSIL).
Para comenzar con algo sencillo, seleccionaremos la plantilla **Aplicación de consola** (en inglés, **Console Application**), modificaremos el nombre del proyecto y su ubicación, si es que no nos gustan las opciones que nos sugiere el entorno, y finalmente presionaremos el botón **OK**.

Hecho esto, el entorno creará un proyecto C# que poseerá dos archivos del tipo **.CS** (por C#, que se pronuncia **C Sharp** y usualmente se abrevia **CS**). Los **.CS** son archivos con código fuente C#; no serán distribuidos con nuestra aplicación, solamente los utilizaremos para crearla.
Si por medio del explorador de soluciones miramos los nombres de los archivos que componen nuestro proyecto, veremos estos dos archivos **.CS**, que se llaman:


```
AssemblyInfo.cs
Class1.cs
```

- **AssemblyInfo**: posee información de nuestra aplicación (nombre del producto, versión, empresa, etc.). Este recurso compilado es situado dentro de nuestra aplicación y se conoce como *assembly*. Luego explicaremos en detalle qué es un *assembly*.
- **Class1**: archivo de código fuente que posee la única clase de la aplicación y el punto de entrada al programa.

No importa lo que esté escrito en dicho archivo, nosotros cambiaremos el programa y escribiremos lo siguiente:

```
class Clase1
{
    static void Main(string[] args)
    {
        System.Console.Write("Nuestro primer programa");
    }
}
```

Analicemos el código anterior. En primer lugar vemos la palabra **class** seguida de un identificador de clase (en este caso, **Clase1**); luego existe un bloque de código encerrado entre llaves.

Quienes tengan algunos mínimos conocimientos de lenguaje C o C++ sabrán que en todo programa debe existir una función llamada **main**, que hace de punto de entrada al programa, es decir, que es la función que debe ejecutarse para iniciar la aplicación.

Habíamos mencionado que en C# todo programa debe estar compuesto de clases, es decir, que no pueden existir funciones globales. Por lo tanto, al menos una clase deberá poseer un método llamado **Main** (esta vez con la "M" en mayúscula), que además deberá ser "estático" (luego veremos qué significa esto).

Dentro del método **Main** se encontrará el código que será ejecutado cuando nuestro programa sea inicializado. En nuestro caso, dicho código será solamente:

```
System.Console.WriteLine("Nuestro primer programa");
```

WriteLine es un método de la clase **Console** que escribe un texto en la salida estándar. Dicho texto se especifica entre comillas (comillas dobles).

¡Bien!, ahora podremos compilar nuestro programa ingresando en el menú **Build**, opción **Build Solution**.

Si hacemos esto, podremos ver que en el panel llamado **Output** (salida) se observa el progreso de la construcción de nuestra aplicación. Si existe algún error, también se mostrará allí como información, y podremos hacer doble clic sobre él para que el entorno nos lleve automáticamente a la línea donde se encontró la falla.

```
—— Build started: Project: ConsoleApplication1, Configuration:
   Debug .NET ——
Preparing resources...
Updating references...
Performing main compilation...
Build complete - 0 errors, 0 warnings
Building satellite assemblies...
———————— Done ————————
    Build: 1 succeeded, 0 failed, 0 skipped
```

Si hemos tipeado todo bien, deberemos ver lo que indica el cuadro anterior, es decir, la notificación de que la aplicación ha sido construida satisfactoriamente.

Luego, podremos ingresar nuevamente en el menú **Build** y seleccionar la opción **Start** (comienzo) o **Start without debugging** (comienzo sin depurar). Lo bueno de esta última opción es que no cierra la ventana donde está nuestro programa cuando éste termina, sino que nos solicita que ingresemos una tecla para continuar.

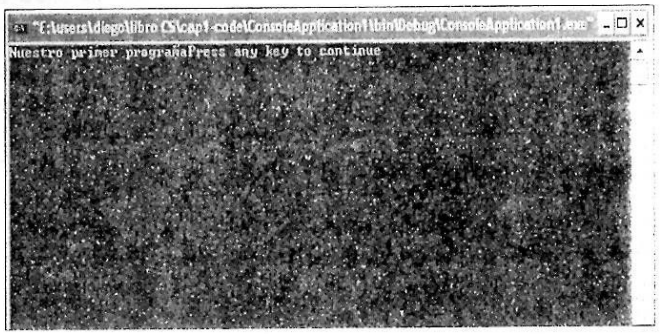


***Figura 8.** Nuestro primer programa C#.*

Claro que si no tenemos Visual Studio a mano, pero hemos instalado el framework .NET, podremos realizar la compilación de modo manual (o para expresarnos mejor: utilizando el compilador desde comando de línea). Al fin y al cabo, habíamos comentado que Visual Studio, entre muchas otras cosas, es un *front end* al compilador C# (y otros compiladores y herramientas). El compilador de C# es una aplicación llamada **csc.exe**, que se encuentra en el directorio de **Windows**, dentro de la carpeta **Microsoft.NET/Framework**, y allí, dentro de la versión del framework que tengamos instalado (por ejemplo, **v1.1.4322**).

Teniendo un archivo llamado, por ejemplo, **programa.cs**, podremos realizar una compilación en comando de línea escribiendo:

```
csc programa.cs
```

Esto nos dejará como salida, en el directorio donde nos encontremos y si no hubo errores en la compilación, un archivo llamado "programa.exe". Si ejecuta la aplicación **csc.exe** con el parámetro **/?**, se listarán los switchs válidos del compilador.

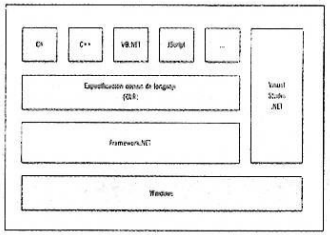
RESUMEN

Está bien, nuestra primera aplicación no ha sido muy excitante, pero al menos hemos dado el primer paso y eso no es poco. También es cierto que algunas cuestiones han quedado sin mucha explicación: ¿qué es una clase?, ¿qué es un método estático?, ¿qué es System? No nos desesperemos; a medida que avancemos en el libro, iremos tratando todos estos temas de manera detallada. Ya hemos visto el conejo blanco, ahora sólo deberemos seguirlo.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué características diferencian al lenguaje C# de C++?



2 ¿Qué es el código MSIL?

3 ¿Qué es la BCL y para que se utiliza?

4 ¿Qué es el CLR?

5 ¿En qué plataformas se pueden ejecutar los programas creados en C#?

6 ¿Es posible compilar un programa C# sin hacer uso del Visual Studio .NET?

PROGRAMACIÓN C#

Capítulo 2

Fundamentos del lenguaje C#

En el capítulo anterior presentamos la plataforma y el lenguaje que trataremos en este libro. Ahora nos sumergiremos en los conceptos fundamentales de C#, lo que será provechoso tanto para quienes den sus primeros pasos en la programación como para quienes vengan de lenguajes como Visual Basic, C++ o Java.

SINTAXIS BÁSICA DEL LENGUAJE

En C# un programa es una colección de clases, aquí no se permitirán funciones ni variables globales como en el caso de C++. C# es un lenguaje orientado a objetos que pretende ser fiel a sus conceptos, y por medio del framework .NET ofrece una librería de clases que nos serán de gran utilidad cuando construyamos aplicaciones.

Esta librería de clases, que mencionamos en el capítulo anterior, se llama **BCL** (*Base Class Library*), y se podría decir que es el sueño de todo programador, ya que ofrece una gran variedad de funcionalidades que iremos estudiando en el desarrollo del libro.

Por el momento, de esta librería diremos que se encuentra organizada por espacios de nombres (en inglés, *namespaces*).

ESPACIOS DE NOMBRES

Los espacios de nombres son un recurso que utilizan algunos lenguajes para la organización de un gran conjunto de tipos de datos. Así como los directorios (o carpetas) permiten organizar archivos de modo que ficheros del mismo nombre convivan en directorios distintos, los espacios de nombres realizan una tarea similar, pero de manera lógica, para tipos de datos extendidos como clases y estructuras.

De este modo, dos clases distintas con el mismo nombre podrían ser utilizadas en el mismo proyecto sin que se suscite ningún error.
Los espacios de nombres también pueden ser anidados, por lo que puede haber unos dentro de otros y, de este modo, mejorar la organización de las librerías de un proyecto.

Gracias a esta facilidad, los nombres de los tipos de datos que nosotros crearemos en el futuro podrán ser de lo más sencillos, sin temor a que ya exista alguna clase o estructura de nombre igual en otra librería que luego podremos querer utilizar.

NOMBRAMIENTO DE FUNCIONES EN LENGUAJE C

El lenguaje C no posee esta facilidad (aunque sí el C++), por lo tanto es normal que una librería deba componer los nombres de las funciones que define, anteponiendo un prefijo –usualmente, una abreviación del nombre de la librería– a cada una de ellas para evitar conflictos con otros paquetes (por ejemplo, todas las funciones de la librería SDL comienzan con **SDL_**).

Veamos cuáles son los espacios de nombres más populares que posee la librería **BCL**:

ESPACIOS DE NOMBRES	DESCRIPCIÓN	EQUIVALENTE EN JAVA
System	Clase base de toda la librería.	Java.lang
System.Collections	Clases e interfaces que definen colecciones de objetos como listas, colas y diccionarios.	java.util
System.Data	Acceso a datos multiproveedor.	java.sql
System.Drawing	Provee acceso a funcionalidades gráficas de GDI+.	java.awt
System.IO	Manejo de entrada y salida a archivos y flujos.	java.io
System.Net	Manejo de comunicación vía red por medio de una gran variedad de protocolos.	java.net
System.Reflection	Provee acceso a información de clases cargadas, por lo que se pueden crear e invocar tipos de modo dinámico.	java.lang.reflect
System.Text	Ofrece recursos para la manipulación de texto en distintas codificaciones.	java.text
System.Threading	Provee recursos para la programación de aplicaciones multihilo.	java.util.concurrent
System.Security	Provee acceso a funcionalidades de encriptación y configuración de seguridad del CLR.	java.security, javax.crypto
System.Windows.Forms	Manejo de ventanas y controles gráficos.	javax.swing
System.Xml	Manejo de datos en formato XML.	javax.xml

***Tabla 1.** Espacios de nombres.*

Por supuesto que la lista es aún más grande. Un acceso de referencia ideal es la documentación que provee Visual Studio .NET o la provista por **MSDN** (*MSDN Library*). Ahora, si volvemos a ver el código de nuestro primer programa creado en el capítulo anterior, entenderemos mejor aquello de **System.Console.WriteLine**. **Console** es una clase que pertenece al espacio de nombres **System**. Si no lo hubiésemos antepuesto al nombre de la clase, el compilador habría arrojado un error, pues no habría encontrado a **Console** en el espacio de nombres global.

La sentencia using

Un modo de ahorrarnos trabajo, al tener que anteponer nombres de *namespaces* a cada una de las clases que utilicemos, es colocar en la cabecera de nuestro archivo de código fuente una sentencia **using**, escribiendo:

```
using <espacio de nombres>;
```

Entonces, el compilador buscará cada clase referenciada en el espacio de nombres global y en los espacios de nombres especificados por sentencias **using**.

Por lo tanto, el código de nuestro primer programa ahora podría ser:

```
using System;
class Clase1
{
      static void Main(string[] args)
      {
            Console.WriteLine("Nuestro primer programa");
      }
}
```

Console es una clase que no existe en el espacio de nombres global, sino en el espacio de nombres **System**. Gracias a que le hemos indicado al compilador que estamos utilizando dicho espacio de nombres, él sabrá que las clases que utilicemos también podrían estar allí.

WriteLine es un método de la clase **Console**. Recordemos que para acceder a información de referencia sobre la BCL podemos utilizar la ayuda que provee Visual Studio, donde veremos todos los métodos y propiedades de la clase **Console**.

LOS COMENTARIOS

En un principio, nuestro programa será pequeño, y analizar qué sucede dentro de él será sencillo, ya que bastará con echar un ojo al código escrito. Sin embargo, a medida que éste crece, se vuelve muy conveniente poder introducir notas aclaratorias sobre qué es lo que pretendemos que el programa haga. Así, si escribimos un algoritmo en particular, podemos agregar un texto que indique qué debe hacer dicho algoritmo, y de este modo otro programador podrá entenderlo rápidamente, o quizá nosotros mismos, releyendo el código que hayamos escrito tiempo atrás.

Para que el compilador no interprete el texto aclaratorio como intentos fallidos de instrucciones, debemos indicarle que estamos introduciendo un comentario.

C# acepta dos tipos de comentarios. Uno es el comentario de línea, que debe comenzar con los caracteres //, como por ejemplo:

```
// Este es un comentario
```


El otro es el comentario clásico del lenguaje C, que comienza con los caracteres /* y termina de manera inversa, con los caracteres */:

```
/* Este es un comentario */
```

Este último tipo de comentario puede ser multilínea; el siguiente es ejemplo de esto:

```
/* Este
es un
comentario*/
```

LAS VARIABLES

Las variables son utilizadas para almacenar datos. Sin embargo, para poder utilizar una variable, en primer lugar deberemos declararla, a diferencia de lo que ocurría en Visual Basic 6.0 o en algunos lenguajes de scripts.

La declaración de una variable está compuesta mínimamente por dos elementos:

- **El tipo de dato.**
- **El identificador.**

El tipo especifica el tamaño en memoria asignado para el dato almacenado por la variable y el correcto modo de presentarlo.
El identificador es el nombre que tendrá la variable y que utilizaremos para referenciarla luego en el programa.

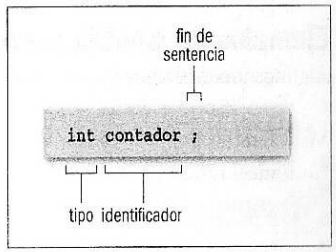


***Figura 1.** Declaración de una variable.*

Para que el compilador los acepte como válidos, los identificadores deberán seguir ciertas reglas, similares a las existentes en otros lenguajes como C, C++, Visual Basic o Java. Veamos:

- Deben comenzar con una letra o un carácter de guión bajo (underscore).
- Deben continuar con una letra, un número o un carácter de guión bajo.
- No deben coincidir con una palabra reservada.

Las palabras reservadas de C#

Todo lenguaje establece un conjunto de palabras reservadas que deberán ser tomadas como prohibidas para la declaración de cualquier tipo de dato definido por el programador. En C# la lista de palabras reservadas es la siguiente:

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

Ejemplos de identificadores

Algunos identificadores válidos son:

MiVariable **_contador**
Contado_ **mi_variable_**

USO DE MAYÚSCULAS Y MINÚSCULAS

Para C# el uso de mayúsculas y minúsculas es importante. Una variable llamada **contador** es distinta de otra llamada **CONTADOR** y de **Contador**. Sin embargo, a pesar de que crear variables diferentes sólo en la utilización de minúsculas y mayúsculas es válido, no es considerado una buena práctica.

Y algunos ejemplos de identificadores no válidos:

- **1Variable**: error porque comienza con un número.
- **$Variable**: error porque comienza son un signo **$**.
- **###Contador###**: error porque comienza con un signo #.
- **default**: error porque **default** es una palabra reservada.

Claro que hay nombres mejores que otros. En un caso real, cuando estemos construyendo nuestras aplicaciones, lo ideal será que el identificador de una variable sea seleccionado en función de un nombre que describa mejor lo que representa dentro del programa. No es una buena idea colocar como identificador de una variable el nombre **Variable**, aunque el compilador lo acepte.

Convenciones de nombramiento

Existen diversas convenciones de nombramiento para clases, variables, funciones, etc. La idea detrás de estas convenciones es especificar reglas para la creación de los identificadores y que, de este modo, especialmente en grupos de trabajo, los nombres asignados sean similares en cuanto a estructura.

Una convención que Microsoft solía recomendar era la notación húngara. A grandes rasgos, en dicha notación se anteponía como prefijo a los nombres de las variables su tipo de dato (codificado en una o dos letras) y una **m_** si era una variable miembro, una **p_** si era un puntero, una **g_** si era una variable global, etc.

Hoy día, esta notación ya no es aconsejada con el mismo fervor, pues los editores modernos rápidamente nos indican el tipo de dato al cual pertenece una variable, y de este modo se pierde una de sus grandes ventajas. Sin embargo, a fin de cuentas, es una cuestión de gustos. Lo importante será mantener la misma convención a lo largo de todo el proyecto.

Este libro adopta la misma notación que utiliza la librería BCL: notación camello para variables, y notación Pascal para clases, estructuras, enumeradores, funciones, métodos, constantes y propiedades.

La notación camello consiste en crear identificadores concatenando palabras sin hacer uso de caracteres especiales (como el underscore o el guión), sino colocando en mayúscula la primera letra de cada palabra, a excepción de la primera.

unaVariableEntera **contador** **nombrePersona**

La notación Pascal es similar a la camello, sólo que la primera letra debe ser colocada en mayúscula.

ImprimirValor **FijarLargo** **DameAltura**

Quienes se encuentren interesados en la notación húngara, en el **Apéndice B** encontrarán su descripción detallada.

Tipos de datos fundamentales

Hemos mencionado que una variable debe poseer un tipo. El lenguaje define un conjunto de tipos de datos primitivos que pueden ser utilizados por nuestras variables, propiedades y métodos. También podremos crear nuestros propios tipos por medio de clases y estructuras, como ya veremos luego.

Una variable tiene asociada una posición de memoria, que es donde realmente se almacena el dato en cuestión. El identificador es, al fin y al cabo, una manera amigable de referirnos a ella. Podríamos decir, entonces, que el tipo de dato es una manera de comunicarle al compilador:

- Cuánto espacio en memoria ocupará la variable.
- Qué semántica tendrá el dato almacenado.

Un número entero podría ocupar en memoria la misma cantidad de espacio que un número de punto flotante. El compilador sabrá cómo interpretar la información que allí encuentre precisamente por el tipo de dato que posee la variable asociada a dicha posición de memoria.

También es conveniente destacar que el compilador distingue entre los tipos de datos por valor y los tipos de datos por referencia. Esto tiene relación con la zona de memoria que se utiliza para crear la variable en cuestión: el *stack* (pila) o el *heap* (montículo). Los tipos de datos por valor serán almacenados en el *stack*, mientras que los tipos de datos por referencia serán almacenados en el *heap* (en el *stack* quedará una referencia de la ubicación en dicha zona de memoria).

Números enteros

C# ofrece tres tipos de números enteros:

- **short**: número entero corto signado.
- **int**: número entero signado.
- **long**: número entero largo signado.

Si ambos son enteros, ¿qué diferencia existe entre ellos? La respuesta es sencilla: la cantidad de memoria que se deberá reservar y cuán grande es el número que cabrá dentro de nuestra variable.

En 2 bytes (short) es posible especificar un número con rango (-32768, 32767), mientras que en 4 bytes (int) es posible especificar un número dentro del rango (-2147483648, 2147483647).

Por otro lado, los tipos de datos numéricos enteros pueden ser signados o no. Si especificamos un número entero corto no signado (ushort), seguimos ocupando 2 bytes, pero al ser todos positivos, el rango válido ahora es (0, 65535). Si intentamos asignar un número negativo a una variable no signada, se producirá una excepción.

Declarando algunas variables enteras:

```
short a;      // Entero corto signado
ushort b;     // Entero corto no signado
int d;        // Entero signado
uint e;       // Entero no signado
long f;       // Entero largo signado
ulong g;      // Entero largo no signado
```

BUSCANDO INFORMACIÓN DE NOTACIONES EN LA RED

Quienes deseen investigar acerca de notaciones en la Red, las siguientes palabras clave arrojarán resultados interesantes: **hungarian notation**, **prefix notation**, **camel notation**, **Pascal notation**.

¿Y LOS PUNTEROS?

Quienes vengan de C o C++ podrían preguntar: ¿dónde están los punteros? C# permite el uso de punteros igual que C++, pero para esto se deberá marcar el bloque o función como **unsafe** (no segura), además de modificar parámetros de compilación. En C# su uso queda relegado a necesidades específicas, como interactuar con recursos no gestionados creados en otros lenguajes.

Diferencias con C/C++

Los lenguajes de programación C/C++ permiten especificar tipos de datos no signados como en C#, sólo que varía el modo de declararlos. En C/C++ se debe especificar el modificador **unsigned** delante del tipo de dato por modificar. Por lo tanto, un entero signado se declararía:

```
//declaración C++ o C#
int a;
```

Mientras que un entero no signado se declararía:

```
//declaración C++
unsigned int a;
//declaración C##
unit a;
```

La idea de reemplazar el **unsigned int** por un simple **uint** tiene que ver con facilitarnos la vida a nosotros, los programadores.
Finalmente, existe también un tipo de dato llamado **byte**, en el cual se puede almacenar, como su nombre lo indica, el valor de un byte. Su rango es (0, 255) y es no signado, aunque es posible declarar un byte signado por medio del tipo de dato **sbyte**.

Números no enteros

C# diferencia en tipo a un número entero de un número con parte fraccionaria; para estos números de datos existen tres tipos: **float**, **double** y **decimal**.
La diferencia entre ellos es la *precisión*. Las computadoras se llevan muy bien con los números enteros, pero para trabajar con números decimales necesitan utilizar un recurso matemático llamado "coma flotante". De esta manera es posible almacenar una cantidad no fija de decimales, pero con una determinada precisión máxima; esta precisión, en definitiva, se encuentra determinada por el tamaño total que ocupa la variable en memoria. Los tres tipos de datos de números no enteros que maneja C# son:

- **float**: número no entero de precisión simple. Las constantes literales deben contener el agregado **f** en la parte decimal para que el compilador pueda tomarlas como float (por ejemplo, **4.0f**).
- **double**: número no entero de precisión doble (por ejemplo, **4.0**).
- **decimal**: número no entero de precisión máxima. Las constantes literales deben contener el agregado **M** en la parte decimal para poder ser tomadas con decimal (por ejemplo, **4.0M**).

Declarando algunas variables no enteras:

```
float a;      // Precisión simple
double b;     // Precisión doble
decimal c;    // Precisión máxima
```

Booleanos

Las variables booleanas pueden albergar dos valores: **true** (*verdadero*) o **false** (*falso*).

Declarando una variable del tipo booleana:

```
bool a;
```

Caracteres

Para almacenar un carácter en una variable debemos especificar su tipo en **char**. Ejemplo:

```
char a;
```

Este tipo de datos puede albergar un carácter del tipo Unicode, pensado especialmente para alfabetos orientales que poseen una cantidad de elementos mayor a 256. Existen ciertos tipos de caracteres "especiales" que son interpretados por el compilador dentro de una cadena de texto. Éstos son conocidos como caracteres de escape, y en la siguiente tabla pueden apreciarse algunos de ellos:

NOMBRE	SECUENCIA DE ESCAPE
línea nueva	\n
tab horizontal	\t
backspace	\b
retorno de carro	\r
barra invertida	\\
comilla doble	\"

***Tabla 2.** Caracteres de escape.*

De este modo, si escribimos en pantalla el siguiente texto:

```
Console.Write("\tTexto1\nTexto2");
```

El primer carácter (\t) será interpretado como un tab horizontal, y el carácter \n, como una línea nueva, por lo que en pantalla podremos observar:

```
    Texto1
Texto2
```

Lista de tipos de datos primitivos

En la siguiente tabla se describen los tipos de datos fundamentales del lenguaje.

TIPO DE DATO FUNDAMENTAL	TAMAÑO EN BYTES	TIPO ESTRUCTURA	DESCRIPCIÓN
bool	1	System.Boolean	Booleano. Su valor puede ser true (verdadero) o false (falso)
sbyte	1	System.SByte	Byte no signado
byte	1	System.Byte	Byte
char	2	System.Char	Carácter
short	2	System.Int16	Entero corto
ushort	2	System.UInt16	Entero corto no signado
int	4	System.Int32	Entero
uint	4	System.UInt32	Entero no signado
long	8	System.Int64	Entero largo
ulong	8	System.UInt64	Entero largo no signado
float	4	System.Single	Número no entero de precisión simple
double	8	System.Double	Número no entero de precisión doble
decimal	16	System.Decimal	Número no entero de máxima precisión

***Tabla 3.** Tipos de datos fundamentales de C#.*

Como se puede observar en la tabla anterior, todo tipo de dato primitivo que define el lenguaje se mapea con una estructura de la librería BCL. Todos los lenguajes que se construyen para la plataforma .NET deberán hacer esta tarea, y de este modo la intercomunicación entre ellos será mucho más sencilla debido a que, en definitiva, utilizan los mismos tipos de datos fundamentales.

Así es que declarar una variable entera como:

```
int a;
```

es exactamente igual a hacerlo del siguiente modo:


```
System.Int32 a;
```

Notemos que el nombre de la estructura relacionada está compuesto por dos partes: el tipo de dato (con la primera letra en mayúscula) y la cantidad de bits que ocupa en memoria. De este modo también podemos inferir que un **int** siempre ocupará 4 bytes, más allá de la plataforma en la cual estemos trabajando (algo que no sucede en C o C++, donde la especificación no ata los tipos de datos a tamaños físicos, sino que simplemente exige una relación entre éstos).

Asignar valor a una variable

Las variables pueden albergar valores, que están relacionados con el tipo de dato de la propiedad en cuestión.

Ya vimos cómo declarar variables, pero no sabemos aún cómo asignarles valor. Para ello deberemos usar el operador de asignación, que en C# es el signo igual (=).

Si declaramos una variable y la utilizamos sin antes inicializarla en algún valor, su contenido será indeterminado.

```
int a = 1;
```

En el caso expuesto hemos fijado el valor 1 a la variable **a** del tipo número entero en la misma línea en que declaramos la variable. Esto podría no ser así; los siguientes también son ejemplos válidos:

```
int a;
// ...
a = 1;
```

¿DÓNDE ESTÁN LOS STRINGS?

C# utiliza una clase que representa los strings y se llama **string** (o **System.String**). Esta implementación es algo distinta del resto de los tipos de datos, ya que éstos son básicamente estructuras. La razón es que en los strings, la cantidad de memoria que se requerirá para la cadena será variable, dependiendo de su contenido. Declarando un **string**: **string miVariableString**.

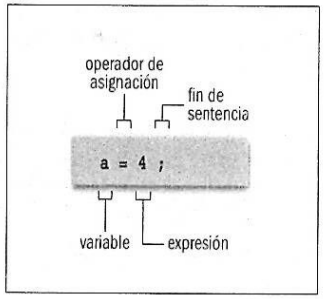


***Figura 2.** El proceso de asignación.*

Constantes

Existen tres constantes:

- **Literales**: especificadas por medio de un valor escrito en el código.

```
int a = 25;
string nombre = "Tito";
```

En el caso citado, **25** es una constante literal al igual que **Tito**.

- **Simbólicas**: especificadas por medio de identificadores asociados a un literal. Son más recomendables que el uso de literales en modo directo, ya que en ciertos contextos éstos podrían ser vistos como números mágicos o sacados de una galera.

Estas constantes son muy similares a las variables, con la diferencia de que a éstas sólo se les podrá dar un valor en el momento de su declaración. Luego, cualquier intento de modificación será tomado como un error por parte del compilador.
Las constantes simbólicas se declaran del mismo modo que las variables, aunque se debe anteponer la sentencia **const** antes del tipo:

```
const int unaConstante = 1;
```

- **Enumeradores**: finalmente, los enumeradores son el tercer tipo de constantes, y adquieren gran utilidad cuando debemos especificar un grupo de constantes estrechamente relacionadas.
Supongamos que deseamos que una variable pueda adoptar el valor de un mes específico del año. Podríamos definir que esta variable fuese numérica entera y

adoptar la convención de que su número equivaliera al número del mes en cuestión. Sin embargo, esta "convención" debería estar especificada en algún documento que se encuentre al alcance de todos los programadores; incluso nosotros, tiempo después de haber escrito el código, podríamos dudar acerca de si el mes de enero era representado por el 1 o acaso por el 0.

Para evitar este tipo de inconvenientes y, también, que algún descuidado especifique un número fuera de rango para el mes (por ejemplo, 13), sin tener que estar colocando código que realice esta verificación, podríamos crear el siguiente enumerador:

```
enum Mes
{
  Enero,
  Febrero,
  Marzo,
  Abril,
  Mayo,
  Junio,
  Julio,
  Agosto,
  Septiembre,
  Octubre,
  Noviembre,
  Diciembre
}
```

Luego, podríamos declarar una variable como del tipo **Mes**. Veamos:

```
Mes m;
```

Ahora, **m** sólo podrá adoptar los valores constantes especificados en el enumerador **Mes**.

```
m = Mes.Enero;
```

Si analizamos cómo declaramos el enumerador, observaremos que éste tiene la palabra reservada **enum** antepuesta a un identificador, y luego existe una lista de constantes separadas por comas dentro del cuerpo declarativo del enumerador que está delimitado por llaves.

Estas constantes podrían poseer un valor literal asignado a ellas por medio del operador de asignación:

```
enum Mes
{
  Enero = 1,
  Febrero = 2,
  Marzo = 3,
  Abril = 4,
  Mayo = 5,
  Junio = 6,
  Julio = 7,
  Agosto = 8,
  Septiembre = 9,
  Octubre = 10,
  Noviembre = 11,
  Diciembre = 12
}
```

En realidad, cada constante dentro de un enumerador siempre posee un valor literal numérico asociado a ella, sólo que si no lo especificamos, lo hará automáticamente el compilador, empezando por el número 0 e incrementando en una unidad. De hecho, podríamos consultar por el valor numérico asociado a una variable de tipo enumerador por medio de conversiones de tipo, como veremos a continuación.

Conversiones de tipo de datos

C# es un lenguaje muy estricto en cuanto a conversiones de tipo. Quienes provengan de otros lenguajes como C o C++ se darán cuenta de esto enseguida, cuando el compilador arroje errores donde antes, utilizando otros lenguajes, arrojaba advertencias. Existen dos tipos de conversiones: **implícitas** y **explícitas**.

MÁS ACERCA DE LOS ENUMERADORES

Dos constantes definidas dentro de un enumerador podrían poseer el mismo valor numérico. Esto sería útil para definir constantes que sean sinónimos; por ejemplo, tanto "setiembre" como "septiembre" podrían estar relacionados con el valor 9.

Las **implícitas** son las que realiza el compilador sin requerir de nosotros la especificación de ninguna sentencia adicional. Este tipo de conversiones se caracteriza por el hecho de que nunca se pierde información ni precisión. La conversión es siempre válida. En el siguiente ejemplo vemos cómo un número entero es convertido a un entero largo. La conversión se da en la segunda línea y es implícita.

```
int n1 = 10;
long n2 = n1;
```

El otro tipo de conversión es **explícito**, y el compilador lo exige cuando podría ocurrir pérdida de información o precisión.

```
long n1 = 10;
int n2 = n1; // error
```

El código anterior arrojará un error en la segunda línea, pues el compilador no puede realizar dicha conversión de modo implícito.
Para realizar una conversión explícita deberemos utilizar una expresión de *casting*. Esto significa especificar el tipo de dato al cual deseamos llevar la expresión, ubicado entre paréntesis y antes de la expresión en cuestión:

```
long n1 = 10;
int n2 = (int) n1; // realizamos un cast
```

Ahora, el compilador ya no protestará. Pero ¿qué sucede si el número en **n1** es más grande de lo que podría albergar **n2**? ¡Se pierde información! Es por esta razón por lo que el compilador nos exige la conversión explícita, ya que es una manera de indicarle que nosotros sabemos lo que hacemos. Veamos el siguiente ejemplo:

```
long n1 = Int64.MaxValue;
int n2 = (int) n1; // realizamos un cast
```

Como es de suponer, en **n2** no quedará el mismo valor que en **n1**, simplemente porque en **n2** no hay lugar para albergar la magnitud que posee **n1**. Sin embargo, el compilador no arrojará error alguno. ¿Cómo decirle al compilador que deseamos ser alertados ante semejante circunstancia? Utilizando un bloque **checked** como se muestra a continuación:

```
long n1 = Int64.MaxValue;

checked
{
    int n2 = (int) n1; // realizamos un cast
}
```

Ahora, si la operación fracasa, el compilador arrojará una excepción. Esta excepción podrá ser capturada y tratada como veremos más adelante.
Y así como existe el bloque **checked** también existe su contrapartida, llamada **unchecked**. ¿Cuándo utilizaremos un bloque **unchecked**? Cuando el compilador no nos permita realizar una operación, a pesar de realizar una conversión de tipos explícita como la siguiente:

```
short n1 = 60000;
```

No hace falta ser un genio para saber que lo escrito en el código anterior fracasará. El rango de un **short** es de (-32768, 32767), por lo que 60000 es un número demasiado grande. Tal vez el ejemplo sea trivial, pero si queremos que dicho código sea compilado de todos modos, entonces deberemos encerrarlo en un bloque **unchecked**:

```
unchecked
{
    short n1 = 60000;
}
```

La clase Convert

La librería **BCL** nos ofrece una clase llamada **Convert** dentro del espacio de nombres **System**, que es muy útil en el momento de realizar conversiones. La gran variedad de métodos estáticos que posee **Convert** nos permite pasar un tipo de dato primitivo a cualquier otro tipo de dato primitivo.
Por ejemplo, la siguiente operación no es válida debido a que intentamos pasar de un número entero a un tipo booleano.

```
bool a = 1;
```

Pero si, en cambio, escribimos:

```
bool a = Convert.ToBoolean(1);
```

Entonces la variable **a**, si recibe un número entero distinto de 0, será **true**, y en caso contrario, **false**.

ERRORES DE CONVERSIÓN

Cuando el compilador no pueda realizar una conversión de tipos de datos, arrojará un error del tipo "Cannot implicity convert type...". Un doble clic sobre dicho error nos llevará a la línea desde la cual proviene el mensaje. Deberemos pensar si hemos cometido algún error o si nos olvidamos de especificar la conversión de modo explícito para sobrescribir el sistema de revisiones del compilador.

Operadores

Es posible combinar números y variables en expresiones por medio de operadores matemáticos. Veamos:

```
int a = 3 + 4;   // Almaceno el número 7 en a
int a = 4;
int b = a / 2;   // Almaceno en b el contenido de la variable
                    dividido por 2
int a  = 10;
int b = a * 14;  // Almaceno en b el contenido de la variable
                    multiplicado por 14
```

También podemos usar tipos de datos numéricos decimales:

```
float a = 3.4f / 2;
```

Y podremos combinar tipos de datos en operaciones:

```
int a = 5;
float b = 4.3f;
float resultado = a * b;
```

Sin embargo, cuando operamos de este modo es conveniente tener mucho cuidado, ya que el tipo de dato resultante estará en función de la operación que realicemos.

Por ejemplo, la suma de dos números enteros dará como resultado un número entero, y la división de dos números enteros también dará como resultado un número entero; esto último es lo que precisamente puede traer muchos problemas.

Si yo le pregunto cuál es el resultado de la operación 5 dividido 2, usted seguramente contestará ¡2,5! Pero nuestra computadora, en cambio, nos devolverá 2, ya que la operación fue entera y el resultado fue de enteros. Colocar una variable de tipo flotante al resultado no soluciona este inconveniente:

```
float a = 5 / 2;    // ¡El resultado de la operación será 2!
```

Para solucionar este inconveniente deberemos convertir al menos uno de los operandos en flotante, para que, de este modo, la división no sea entre números enteros, sino entre un entero y un número flotante:

```
float a = 5.0f / 2;          // Ahora el resultado será 2.5
```

Uso de constantes en operaciones

Utilizar constantes en lugar de números literales de manera directa es considerado una buena práctica, ya que facilita la lectura y comprensión del código fuente. Por ejemplo:

Constantes numéricas literales:
superficie = 3.1415f * 16;

Constantes declaradas:
const float pi = 3.1415f;
superficie = pi * (radio*radio);

NÚMEROS MÁGICOS

Colocar una constante literal escrita sobre el código sin explicación alguna es una mala práctica que suele denominarse "números mágicos". Existe un muy buen libro llamado **Code Complete**, de Steve McConnell, que cita muchas de las malas prácticas más populares e indica cómo evitarlas. Lectura obligada si se desea ser un buen programador.

El último listado de código es un poco más legible. Más allá de lo trivial del ejemplo, en otros casos podría no resultar obvio, en el análisis de una expresión, que representan determinados números. En cambio, al ver una constante llamada **pi** rápidamente entenderemos a qué nos referimos.

CONTROL DEL FLUJO DE EJECUCIÓN

Si especificamos un programa como un simple conjunto de operaciones sobre variables e invocaciones a métodos de clases, su ejecución será secuencial desde la primera línea hasta la última. Este escenario no es muy emocionante, y dista mucho de las necesidades que poseen los programadores para la creación de aplicaciones del mundo real.

Para esto el lenguaje C# ofrece un conjunto de sentencias que permiten controlar el flujo de ejecución, es decir, qué línea de código será ejecutada por el procesador.

De este modo podremos indicarle a nuestro compilador que una determinada secuencia de instrucciones no debe ejecutarse sólo una vez sino diez, o que una porción de código debe ejecutarse sólo si una variable es igual a 16 o, quizás, que repita una porción de código hasta que el usuario presione una tecla determinada.

Sentencias condicionales

La primera sentencia que veremos será la condicional **if**. Con esta sentencia podremos expresar, por ejemplo, que:

Si (<expresión booleana>) entonces hacer <sentencia> [Si no, hacer <sentencia>]

La expresión usualmente es una comparación utilizando los operadores:

OPERADOR	SIGNIFICADO
==	Igual a
!=	Distinto a
<	Menor a
>	Mayor a
<=	Menor o igual a
>=	Mayor o igual a

***Tabla 4.** Operadores de comparación.*

Veamos un ejemplo:

```
int a = Console.In.Peek();
if (a == 'x')
    Console.Write("Has adivinado el caracter secreto");
```

En la primera línea utilizamos el método **Peek** para tomar un carácter desde la entrada de consola. Luego verificamos si este carácter es igual a un carácter arbitrario. Entonces **a == 'x'** es una expresión que se evalúa por verdadero o falso.
Si la expresión es verdadera, ingresamos en el cuerpo de la sentencia condicional; el cuerpo siempre será la sentencia próxima al cierre del paréntesis que simboliza el fin de la expresión booleana. Es posible que nuestra sentencia sea, en realidad, un grupo de sentencias; en estos casos deberemos especificar un bloque de código con los caracteres **llaves** ({ }); la llave abierta indica el comienzo de un bloque de código, mientras que la llave cerrada indica el fin del bloque. Veamos:

```
int a = Console.In.Peek();
if (a == 'x')
{
    Console.Write("Has adivinado el carácter secreto");
    Console.Write("Felicitaciones");
}
```

Otros lenguajes como Visual Basic utilizan otros criterios para exigir una sentencia de fin de condicional, como **End If** o similares. C#, como C y C++, utiliza el concepto de bloques en todas sus sentencias. De hecho, las variables declaradas dentro de bloques serán locales a éstos.
¿Y qué ocurre cuando la expresión booleana especificada en el **if** se evalúa por falso? Bueno, una sentencia condicional **if** debe poseer una sentencia que es el código que se ejecuta ante una evaluación por verdadero de la expresión booleana, y puede o no

EL USO DE LAS SANGRÍAS

Es muy recomendable el uso de sangrías para mejorar la legibilidad de nuestro código. Cuando escribimos el código del cuerpo de las sentencias **if**, lo hacemos dejando una sangría de una determinada cantidad de espacios (usualmente cuatro). En los entornos de desarrollo integrado y en casi cualquier editor de textos, lo hacemos por medio de la tecla **TAB**.

poseer una sentencia por ejecutar si la expresión es evaluada por falso (esta sentencia debe ser precedida por la palabra clave **else**). Entonces, los escenarios posibles son dos:

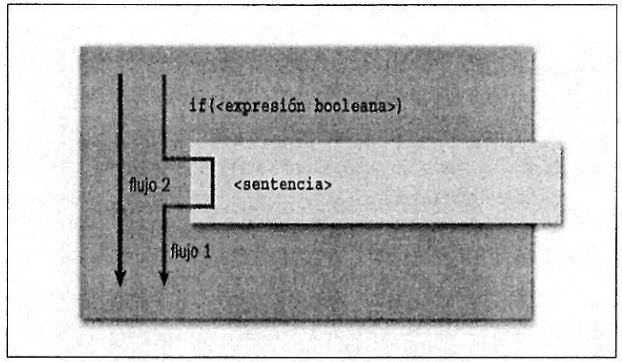


***Figura 3.** Sentencia condicional sin else.*

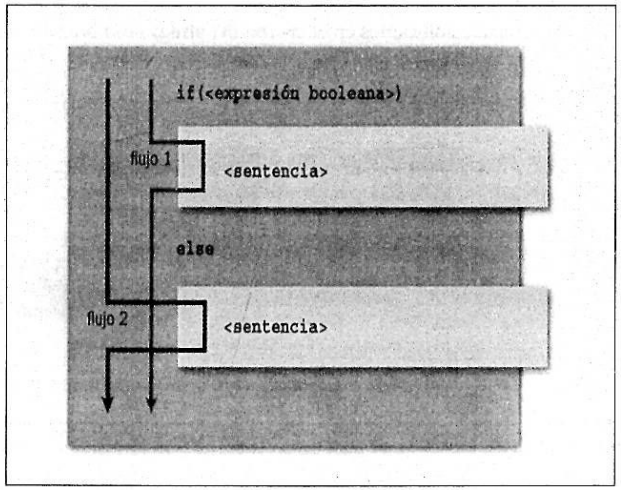


***Figura 4.** Sentencia condicional con else.*

¿QUÉ TIPO DE SENTENCIAS PUEDEN ESCRIBIRSE?

El código escrito en el cuerpo de un **if** no está limitado en ningún sentido. Allí podremos declarar variables, efectuar asignaciones, especificar otras sentencias, invocar funciones, etc.

Como se puede apreciar en las **Figuras 3** y **4**, el flujo 1 corresponde a una evaluación verdadera de la expresión, mientras que el flujo 2 corresponde a una evaluación por falso. Veamos el ejemplo de código especificando la sentencia **else**:

```
int a = Console.In.Peek();
if (a == 'x')
{
     Console.Write("Has adivinado el carácter secreto");
     Console.Write("Felicitaciones");
}
else
{
     Console.Write("Tal vez deberías intentarlo de nuevo");
}
```

Nótese que las llaves que colocamos en el cuerpo del **else** no son obligatorias en este caso, pues éste posee sólo una línea.

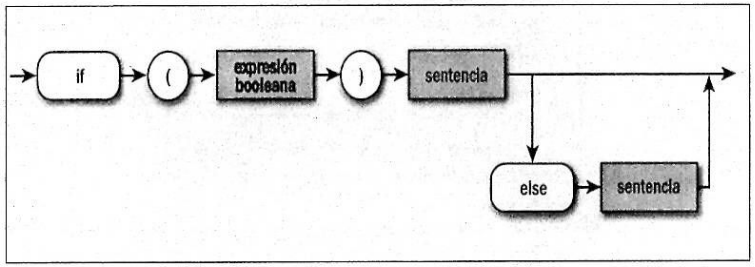


***Figura 5.** Sintaxis de if.*

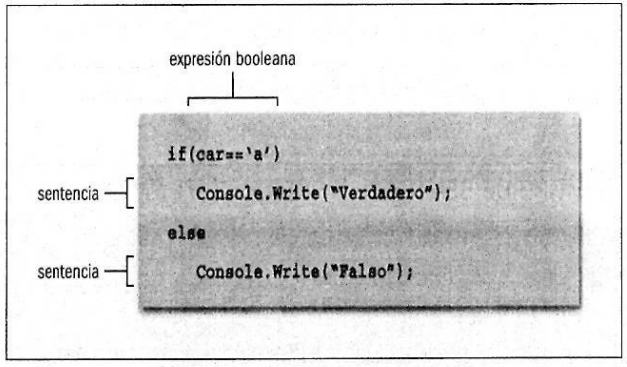


***Figura 6.** Ejemplo de la sentencia if.*

¿Y qué ocurre si las opciones son múltiples? Si, por ejemplo, deseamos solicitar un número del 1 al 3 y realizar diferentes acciones en función de la opción elegida.

Bueno, nada nos impide escribir lo siguiente:

```
if (num == 1)
{
     // opción 1
}
else
{
     if (num == 2)
     {
          // opción 2
     }
     else
     {
          if (num == 3)
          {
               // opción 3
          }
          else
          {
               // ninguna opción
          }
     }
}
```

En el listado anterior hemos anidado sentencias condicionales **if**. Analicemos cómo se ejecuta dicho código.

OPERADOR DE IGUALDAD

Nótese que el operador de comparación de igualdad es "==" y no "=" (que es el operador de asignación). Esto, en un principio, podría confundirnos, sobre todo si venimos de otros lenguajes. Afortunadamente, si nos equivocamos y utilizamos "=" en lugar de "==", el compilador nos lo informará, ya que la expresión condicional debe poder ser evaluada por verdadero o por falso.

Supongamos que **num** es igual a **3**.

1. Se evalúa **num == 1**. Como **3** es distinto de **1**, entonces la expresión se evalúa por falso y el flujo de ejecución ingresa en el cuerpo **else** de la sentencia.
2. Dentro del **else** del primer **if** se encuentra otro **if** (esto es algo totalmente válido); se evalúa la expresión **num == 2** como falsa, ya que **3** es distinto de **2**. Nuevamente se salta al **else** del **if**, en este caso, del segundo.
3. Dentro del segundo **else** nos encontramos con otro **if** con una expresión que evalúa **num == 3** por verdadero, ya que **3** es igual a **3**. Entonces, el flujo de ejecución ingresa en el cuerpo que posee el comentario **opción 3**.

Pero existe una manera de escribir lo mismo que mejora mucho la legibilidad.

```
if (a == '1')
{
    // opción 1
}
else if (a == '2')
{
    // opción 2
}
else if (a == '3')
{
    // opción 3
}
else
{
    // ninguna opción
}
```

En esta situación, la idea es exactamente la misma. De hecho, el compilador no encontrará diferencia alguna. Únicamente hemos eliminado las llaves de los bloques **else**, ya que dentro de éstos existe sólo una sentencia **if** (no tiene importancia que a su vez esta sentencia posea más de una línea u otras sentencias dentro), y hemos modificado el uso de las sangrías.

Operadores lógicos

Dentro de los paréntesis del **if** debemos colocar una expresión booleana. Dicha expresión puede ser trivial, como, por ejemplo:


```
bool salir = true;
if (salir)
    Salir_del_programa
```

En el ejemplo citado, la expresión sólo consiste en una variable. Esta expresión, como cualquier otra, será evaluada por verdadero o falso.

Pero ¿qué sucede si queremos ejecutar ciertas instrucciones sólo si **a** es igual a **1** y **b** es igual a **2**? Entonces podremos crear una expresión del siguiente modo:

```
if (a == 1 && b == 2)
    Hacer_algo
```

El operador **&&** es el operador lógico **AND**, por lo que nuestra expresión fue equivalente a decir: **"Si a es igual a 1 Y b es igual a 2, entonces..."**.

Por lo tanto, sólo ingresaríamos en el cuerpo de la sentencia **if** cuando la expresión **a == 1** y la expresión **b == 2** se evaluaran por verdadero.
También podríamos desear realizar una determinada acción cuando **a** fuera igual a **1** o **b** fuera igual a **2**. En ese caso utilizaríamos el operador lógico **OR** (||):

```
if (a == 1 || b == 2)
    Hacer_algo
```

De este modo podríamos crear expresiones aún más complejas combinando operadores lógicos a nuestro gusto, pero siempre deberemos tener como meta la creación de un código lo más legible posible, ya que, recordemos, éste no será leído sólo por una computadora.

OPERADORES	DESCRIPCIÓN
&&	Y lógico (conjunción)
||	O lógico (disyunción)
!	Negación lógica

***Tabla 5.** Operadores lógicos.*

La sentencia switch

La sentencia **switch** nos permite modificar el flujo de ejecución en función de la evaluación de una expresión. Es ideal para ciertos casos; por ejemplo, cuando una

variable puede tomar un valor de un conjunto de valores conocidos y deseamos actuar en consecuencia. Podría reemplazar el uso de sentencias **if** anidadas como en el ejemplo visto anteriormente.

```
switch (<expresión>)
{
    case <expresión constante> : [<sentencia>] [break;]
    case <expresión constante> : [<sentencia>] [break;]
    case <expresión constante> : [<sentencia>] [break;]
    [default: [<sentencia>] break;]
}
```

Suponiendo que **num** sea una variable de tipo entera, veamos el siguiente ejemplo:

```
switch (num)
{
    case 1:
        Console.Write("Opción 1");
        break;
    case 2:
        Console.Write("Opción 2");
        break;
    case 3:
        Console.Write("Opción 3");
        break;
    default:
        Console.Write("Ninguna opción");
        break;
}
```

Como podemos apreciar en el listado anterior, **num** se coloca como expresión para evaluar por el **switch**. Luego, en cada **case** colocaremos los valores posibles de esta variable que deseemos tratar. El **break** impide que el flujo de ejecución salte de un **case** al siguiente, aunque en realidad esta exigencia es heredada del uso en otros lenguajes como C, C++ o Java, ya que, de todos modos, en C# no se permite el salto silencioso de un **case** a otro.

Si quisiéramos ingresar en el **case 1**, por ejemplo, y saltar de aquí al siguiente (el **case 2**), deberíamos escribir:


```
switch (num)
{
    case 1:
        Console.Write("Opción 1");
        goto case 2;
    case 2:
        Console.Write("Opción 2");
        break;
    case 3:
        Console.Write("Opción 3");
        break;
    default:
        Console.Write("Ninguna opción");
        break;
}
```

En el caso de que el tratamiento de una opción sea igual al de otra, podremos escribir:

```
switch (num)
{
    case 1:
    case 2:
        Console.Write("Opción 1 o 2");
        break;
    case 3:
        Console.Write("Opción 3");
        break;
    default:
        Console.Write("Ninguna opción");
        break;
}
```

En el caso de que ningún valor de **num** coincida con los **case** que hemos especificado, lo que podemos hacer es colocar un default que funcionaría como una especie de **else** final a toda la sentencia, y sólo se ingresará aquí si no se ingresa en ningún **case**. El **default** no tiene por qué ir al final de la lista de todos los **case**.

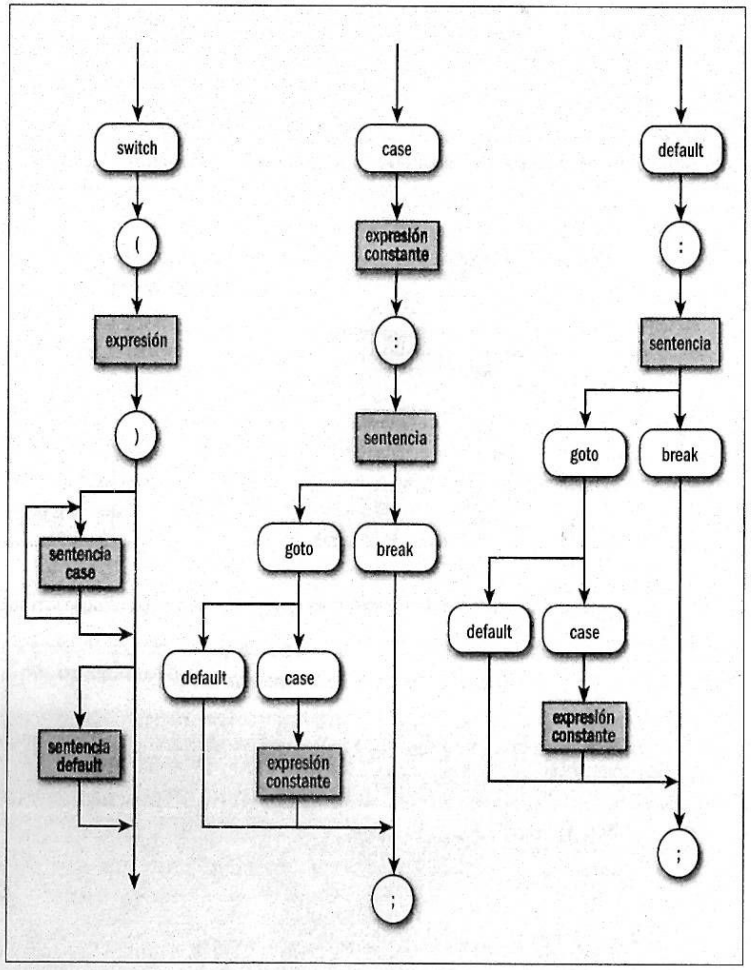


***Figura 7.** Sintaxis de switch, case y default.*

III CUÁNDO UTILIZAR UN SWITCH

La sentencia **switch** suele utilizarse cuando una variable sólo puede adoptar un número establecido de valores constantes y para cada valor se debe realizar un tratamiento diferente.

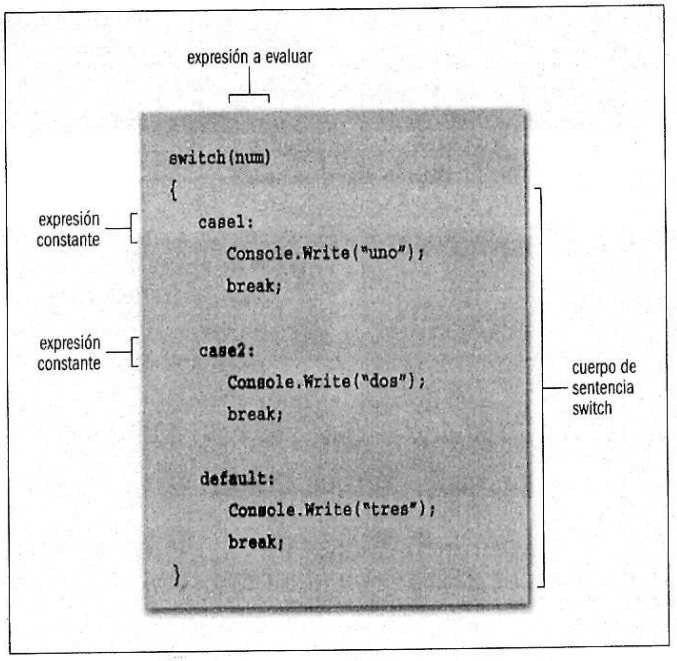


***Figura 8.** Ejemplo de la sentencia switch, case y default.*

Equivalencias entre las sentencias condicionales if y switch

El uso de la sentencia **if** o **switch** es, en ciertos casos, una cuestión de gustos, aunque la verdad es que con un **if** se podrá hacer todo lo que con un **switch**, pero el caso inverso no es cierto. Veamos algunas analogías:

Ejemplo 1:

```
switch (num)
{
    case 1:
    case 2:
    case 3:
        Console.Write("uno,
            dos o tres");
        break;
    case 4:
    case 5:
        Console.Write("cuatro
```

```
if (num >= 1 && num <= 3)
   Console.Write("uno, dos o tres");
else if (num == 4 || num == 5)
   Console.Write("cuatro o cinco");
```

```
        o cinco");
        break;
}
```

Ejemplo 2:

```
switch (num)
{
    case 1:
        Console.Write("uno");
        break;
    case 2:
        Console.Write("dos");
        break;
    default:
        Console.Write("otro");
        break;
}
```

```
if (num == 1)
    Console.Write("uno");
else if (num == 2)
    Console.Write("dos");
else
    Console.Write("otro");
```

Sentencia de bucle

En algunas ocasiones requeriremos que determinada porción del código escrito sea repetida una cierta cantidad de veces o hasta que se cumpla una determinada condición. C# cuenta con varias sentencias para cumplir este fin.

La sentencia while

La sentencia **while** (que significa **mientras**) indica que una sentencia o grupo de sentencias debe repetirse mientras una expresión asociada sea verdadera.

mientras (<expresión booleana>) hacer <sentencia>

LA CONDICIÓN DE TERMINACIÓN DEL BUCLE

La expresión booleana del bucle "mientras" debería ser verdadera en ciertas circunstancias y luego, en algún momento, tornarse falsa. Si la expresión NUNCA fuese verdadera, jamás se ingresaría en el cuerpo del bucle. Si la expresión SIEMPRE fuese verdadera, nuestro programa jamás se terminaría, ya que nunca saldríamos del bucle.

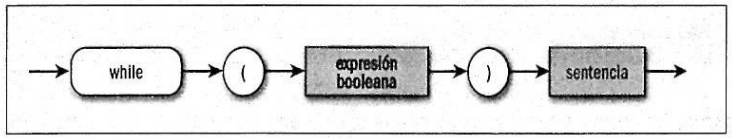


***Figura 9.** Sintaxis de while.*

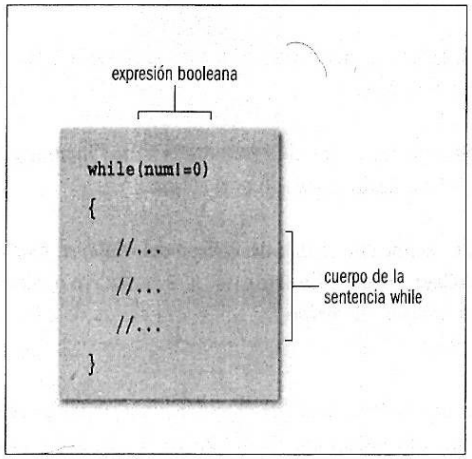


***Figura 10.** Ejemplo de la sentencia while.*

Supongamos que queremos escribir un programa que, mientras el usuario no ingrese un número determinado, no termine:

```
string palabraClave = "";
while (palabraClave != "salir")
{
    Console.WriteLine("Ingrese la palabra clave:");
    palabraClave = Console.In.ReadLine();
}
Console.WriteLine("Correcto");
```

Analicemos el código anterior:

En la primera línea declaramos una variable de tipo cadena y la inicializamos. Esta variable contendrá el texto que se ingrese por teclado.
Inmediatamente después se evalúa la expresión encerrada entre paréntesis de la sentencia **while**; en este caso podríamos traducir la línea

```
while (palabraClave != "salir")
```

como: "mientras la variable **palabraClave** sea distinta de **"salir"**"... Si la expresión es evaluada por verdadero, ingresaremos en el cuerpo del **while**; de otro modo, saltaremos a la línea inmediatamente después del cuerpo de éste.

En este caso una cadena vacía es distinta de la palabra **"salir"**; por lo tanto, ingresamos en el cuerpo del **while**.

Dentro del **while** enviamos a la salida estándar el texto **"Ingrese la palabra clave:"** y volvemos a capturar el texto ingresado por teclado.

Terminada la última línea de código del cuerpo del **while**, se vuelve a evaluar la expresión **"palabraClave != salir"**. En este caso, si es verdadero o falso dependerá de lo ingresado por el usuario. El cuerpo del **while** será ejecutado hasta que la expresión se evalúe como falsa.

Nótese que la expresión booleana que se evalúa en el **while** puede ser, así como vimos con el **if**, una expresión formada de diversas comparaciones conectadas entre sí por operadores lógicos.

```
while (a != 3 && b == 20 && c < 10)
    // ...
```

Esto se traduce como **"mientras a sea distinto de 3 Y b sea igual a 20 Y c sea menor de 10..."**.

La sentencia do

Con el **while**, el bloque de código por repetir no será ejecutado nunca si la expresión es falsa la primera vez. En ciertas ocasiones nos vendría bien ejecutar dicho código al menos una vez, y luego evaluar la expresión.
Para esto existe otra sentencia llamada **do**:

hacer <sentencia> mientras (<expresión booleana>);

Como podemos apreciar, la sentencia **do** es muy similar a la sentencia **while**, sólo que cambió el momento en el cual se evalúa la expresión.

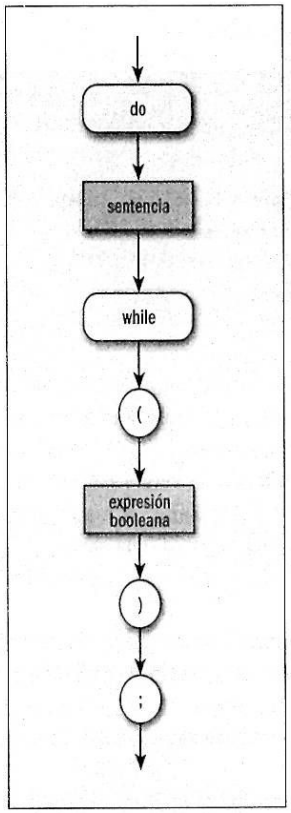


Figura 11. *Sintaxis dowhile.*

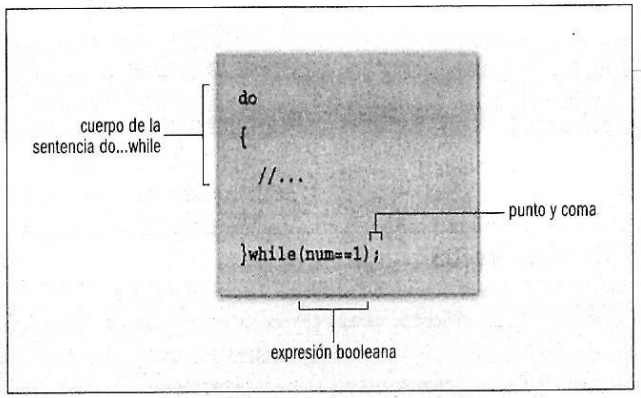


Figura 12. *Ejemplo de la sentencia dowhile.*

Veamos un ejemplo:

```
string palabraClave;
do
{
    Console.Write("Ingrese la palabra clave\n");
    palabraClave = Console.In.ReadLine();
} while (palabraClave != "salir");
Console.Write("Correcto");
```

El programa es muy similar al anterior, aunque esta vez no necesitamos inicializar la variable a un valor conocido, ya que ésta es sobrescrita antes de evaluar la expresión por primera vez.
Entonces, el **"Ingrese la palabra..."** siempre será ejecutado al menos una vez. En el programa anterior (del uso del **while**), si la variable **palabraClave** fuese inicializada en **"salir"**, el bloque interno a él nunca sería ejecutado.

La sentencia for

En ciertas ocasiones tendremos la necesidad de ejecutar un bloque de código un número determinado de veces. C# ofrece para estos casos una sentencia llamada **for**.

para (<sentencia>; <expresión booleana>; <sentencia>) <sentencia a repetir>

La primera sentencia sólo se ejecuta una vez. Luego, la expresión se evalúa antes de ingresar en cada bucle, y la segunda sentencia se ejecuta al finalizar cada bucle. Veamos un ejemplo:

```
for (int i=0; i<5; i++)
    Console.WriteLine("...");
```

BUCLES INFINITOS

En todo programa en que se utilicen sentencias de repetición, existe la probabilidad de que se ingrese en bucles infinitos. Esto significa, por ejemplo, que si la expresión de un **while** siempre fuese verdadera, el **while** nunca acabaría y, por lo tanto, nuestro programa tampoco.

La primera sentencia es:

```
int i=0
```

Aquí declaramos e inicializamos una variable que sólo será válida dentro del cuerpo del **for** (una vez terminado éste, dicha variable no existirá más, es local a la sentencia al bloque del **for**). Esta sentencia sólo se ejecutará la primera vez que ingresemos en el ciclo.

```
i<5
```

Mientras dicha expresión booleana sea válida, se ejecutará el código especificado en el cuerpo del **for**.

```
i++
```

Sentencia que se ejecuta en cada ciclo, incrementa en uno el valor de la variable **i**. Indefectiblemente dicha variable llegará a 5; por lo tanto, la expresión **"i<5"** se evaluará a falsa y el **for** habrá terminado. Por ese motivo, el cuerpo del **for**, en este caso, se ejecutará cinco veces del siguiente modo:

- Se ejecuta la sentencia **"int i=0"** (esta sentencia sólo se ejecuta una vez).
- Se evalúa la expresión **"i<5"**. En este caso **"0<5"** es verdadero, por lo tanto se ingresa en el cuerpo del **for**.
  bucle 0: se ejecuta **Console.WriteLine()**.
  bucle 0: se ejecuta la sentencia **"i++"**. En este caso **i** se incrementa en **1** y pasa a valor **1**.
- Se evalúa la expresión **"i<5"**. En este caso **"1<5"** es verdadero, por lo tanto se ingresa en el cuerpo del **for**.
  bucle 1: se ejecuta **Console.WriteLine ()**.
  bucle 1: se ejecuta la sentencia **"i++"**. En este caso **i** se incrementa en **1** y pasa a valor **2**.
- Se evalúa la expresión **"i<5"**. En este caso **"2<5"** es verdadero, por lo tanto se ingresa en el cuerpo del **for**.
  bucle 2: se ejecuta **Console.WriteLine ()**.
  bucle 2: se ejecuta la sentencia **"i++"**. Por tanto, **i** aumenta en **1** y pasa a valor **3**.
- Se evalúa la expresión **"i<5"**. En este caso **"3<5"** es verdadero, por lo tanto se ingresa en el cuerpo del **for**.

bucle 3: se ejecuta **Console.WriteLine ()**.
bucle 3: se ejecuta la sentencia **"i++"**. En este caso **i** suma **1** y pasa a valor **4**.

- Se evalúa la expresión **"i<5"**. En este caso **"4<5"** es verdadero, por lo tanto se ingresa en el cuerpo del **for**.
  bucle 4: se ejecuta **Console.WriteLine ()**.
  bucle 4: se ejecuta la sentencia **"i++"**. En este caso, **i** suma **1** y pasa a valor **5**.
- Se evalúa la expresión **"i<5"**. En este caso **"5<5"** es falso, por lo que se sale del bucle.

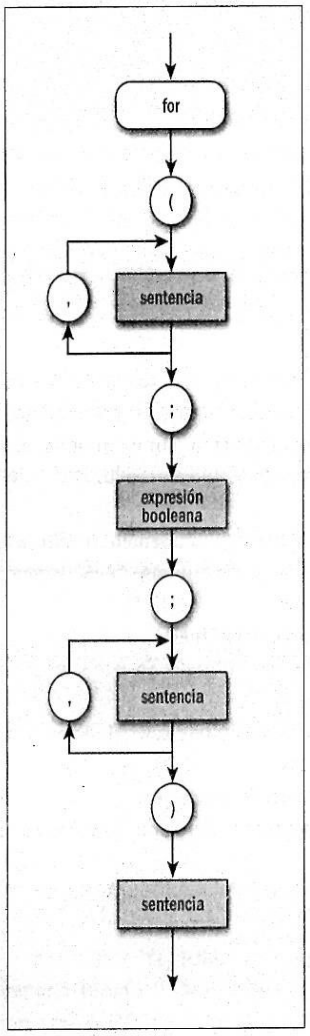


Figura 13. *Sintaxis de For.*

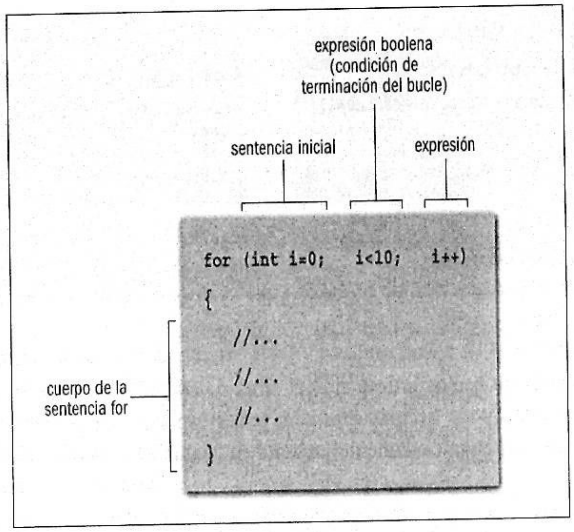


Figura 14. *Ejemplo de la sentencia for.*

Equivalencias entre el for y el while

Como ocurre entre el **if** y el **switch**, donde en ciertos casos el uso de una u otra sentencia es una cuestión de gustos, con el **for** y el **while** ocurre lo mismo.

Veamos ahora algunos ejemplos de código que controlan el flujo de programa del mismo modo, a pesar de utilizar distintas sentencias:

Ejemplo 1: en este caso sería más lógico utilizar el **for**.

```
for (int i=0; i<5; i++)
Console.Write("...\n");
```

```
int i=0;
while (i<5)
{
    Console.Write("...\n");
    i++;
}
```

Ejemplo 2: en este caso sería más lógico utilizar el **while**.

```
string palabraClave = "";
for (; palabraClave != "salir";)
```

```
String palabraClave = "";
While (palabraClave != "salir")
```

```
{
  Console.Write("."); 
  PalabraClave = Console.ReadLine();

}
```

```
{
  Console.Write("."); 
  palabraClave = Console.ReadLine();

}
```

La sentencia foreach

Existe una última sentencia de repetición que es muy útil para trabajar con colecciones de datos.

Si bien todavía no hemos introducido el tema de colecciones ni el de arrays, nos adelantaremos un poco simplemente para poder explicar esta poderosa sentencia que se encuentra (desgraciadamente) ausente en lenguajes como C o C++.

Una colección es un conjunto de elementos, en general, del mismo tipo, aunque no necesariamente. Utilizando algún tipo de colección se puede definir un conjunto de *n* elementos, con la posibilidad de acceder a cada uno de ellos por medio de un subíndice. Veamos cómo declararlo haciendo uso de un array clásico:

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

Ahora podríamos acceder al primer elemento utilizando el operador de indexación **[]**:

```
enteros[0] = 100;
```

Y al segundo elemento:

```
enteros[1] = 500;
```

Y al tercero (y último):

```
enteros[2] = 900;
```

Nótese que el acceso a elementos del array se encuentra basado en cero (y no en uno, como en otros lenguajes).

Trabajar con arrays puede ser muy cómodo, pero no lo sería tanto si el lenguaje no ofreciese sentencias adecuadas para manipularlos. La sentencia **for** es una de ellas. Supongamos que deseamos inicializar los elementos de un array en un determinado número; un modo de hacerlo sería escribiendo:

```
int[] enteros = new int[3];
for (int i=0; i<enteros.Length; i++)
    enteros[i] = 0;
```

Si modificamos el tamaño de nuestro array y pasamos de 3 a 300 elementos, la sentencia **for** permanecerá inalterada.
Pero la sentencia **for** no es la única que ofrece el lenguaje para este tipo de casos. Existe otra llamada **foreach** que permite manipular arrays de una manera más amigable, si es que esto es posible.

Por cada (<tipo> <elemento> en <colección>) <sentencia>

Veamos cómo iterar por todos los elementos de una colección para visualizar su contenido en pantalla:

```
foreach (int num in enteros)
    Console.Write("número {0}\n", num);
```

Dentro del paréntesis del **foreach** primero declaramos una variable que oficiará de elemento activo de la colección; ésta es seguida de la palabra clave **in** y, finalmente, la colección por la cual deseamos iterar.
Entonces, en cada iteración, en lugar de tener que acceder al elemento por subíndice, directamente hacemos uso de la variable "elemento" declarada. Sólo cabe la salvedad de que el uso de la variable **elemento** es sólo para lectura, no podremos practicar una asignación con ella.

{} ¿QUÉ ES LA PROPIEDAD LENGTH?

Length es una propiedad de los arrays que nos indica la cantidad de elementos que éste posee. Más adelante, cuando profundicemos sobre este tema, analizaremos los métodos y propiedades para trabajar con ellos.

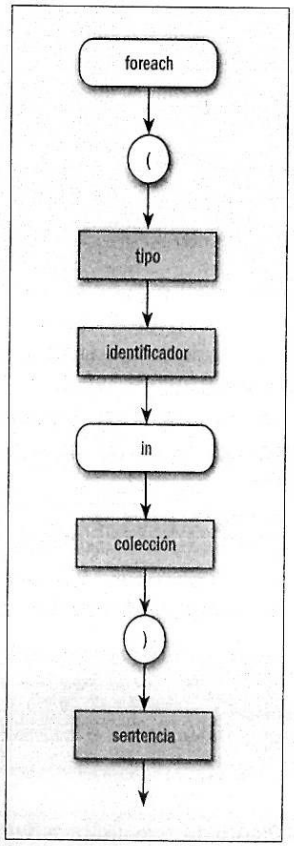


Figura 15. *Sintaxis de foreach.*

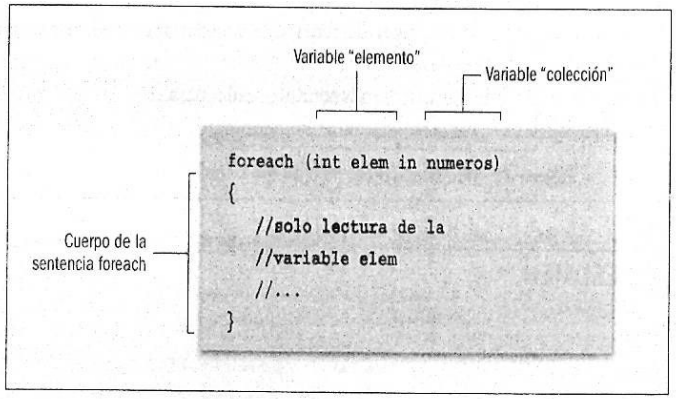


Figura 16. *Ejemplo de la sentencia foreach.*

Tal vez en este momento no puedan apreciarse las ventajas del uso práctico del **foreach** por sobre la sentencia **for**.
Cuando analicemos los distintos tipos de colecciones veremos que resulta más cómodo acceder a ciertos grupos de elementos de este modo, debido a que no siempre es posible acceder a elementos por medio de subíndices como es usual hacer dentro del cuerpo de una sentencia **for**.

La sentencia break y continue

Hemos visto cómo se utiliza la sentencia **break** dentro de sentencias **swith/case**. No obstante ello, el **break** puede ser usado con diversas finalidades. Se utiliza para salir del bucle en el cual nos encontramos, independientemente de la expresión que se evalúa iteración tras iteración.

En numerosas oportunidades, su uso no es considerado una buena práctica, pero deberemos tenerlo en cuenta como una herramienta más que podremos utilizar en el futuro, en caso de que nos haga falta. A continuación, observamos un sencillo ejemplo del uso del **break**:

```
int i = 0;
while (i < 10)
{
      Console.WriteLine("punto a");
      i++;

      if (i == 3)
            break;
}
```

Cuando "`i == 3`" se evalúa como verdadera, salimos del bucle **while** y seguimos con la próxima línea posterior a él (de la misma manera en que habría ocurrido si la expresión "`i < 10`" hubiese sido falsa).

QUE EXISTA NO SIGNIFICA QUE DEBAMOS UTILIZARLO

Siempre se debe dar por terminado un bucle por la evaluación por falso de su expresión booleana. Utilizar la sentencia break suele ser considerado una mala práctica porque altera el flujo de ejecución normal que debería realizar el bucle, lo que empobrece la legibilidad de nuestro programa y aumenta las probabilidades de cometer un error.

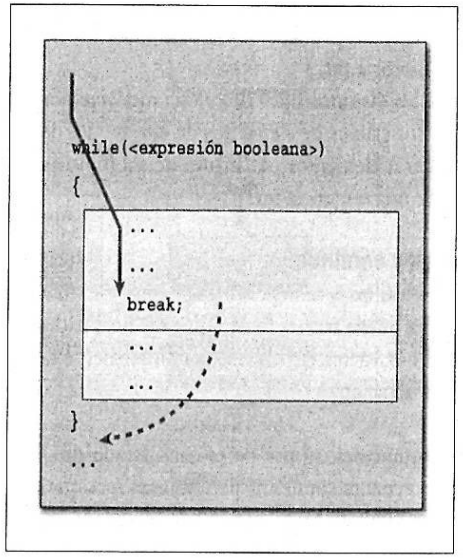


Figura 17. *Modificación del flujo de ejecución ante un break.*

También puede ser aplicada en sentencias **for**:

```
for (int i=0; i<10; i++)
{
      Console.WriteLine("punto a");
      if (i > 3)
            break;
}
```

La sentencia **continue**, en cambio, nos permite saltearnos lo que resta de una iteración dentro de un bucle para seguir con la próxima (sin salir del bucle). Veamos un ejemplo con un **while** y con un **for**:

```
int i = 0;
while (i < 10)
{
      Console.WriteLine("punto a");
      i++;
      if (i > 3)
            continue;
      Console.WriteLine("punto b");
}
```

```
for (int i=0; i<10; i++)
{
      Console.WriteLine("punto a");
      if (i > 3)
            continue;
      Console.WriteLine("punto b");
}
```


Llegado el momento de ejecutar el **continue**, las instrucciones que resten de la iteración en curso serán salteadas y se dará comienzo a la próxima iteración.

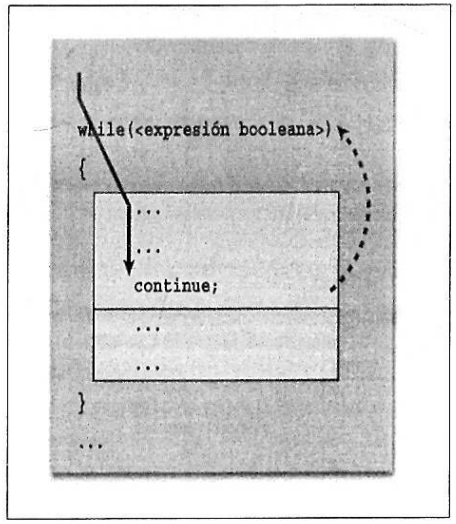


Figura 18. *Modificación del flujo de ejecución ante un continue.*

RESUMEN

Por el momento la programación con C# parece ser bastante similar a C++ o a Java; cuando comencemos a analizar detenidamente las clases que ofrece la librería BCL junto con las facilidades del lenguaje, veremos que estamos ante un lenguaje de programación único.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Dónde se encuentra el error en los siguientes listados?

a)

```
int n1 = 10;
double n2 = n1;
float n3 = n2;
```

b)

```
int valor;
if (valor == 0)
   Console.Write("el valor
   es cero!");
```

c)

```
for (int i=0, i<5, i++)
   Console.Write(".");
```

d)

```
int num = 0;
if (num == 0)
   Console.Write("Es valor");
   Console.Write("es cero!");
else
   Console.Write("El valor
   es distinto de cero");
```

2 ¿Para qué se utilizan los espacios de nombres? ¿Qué función cumplen?

3 ¿Cómo se puede convertir un string que posea un número en su interior a una variable numérica? ¿Es una conversión segura en todos los casos?

Capítulo 3

Clases y objetos

En este capítulo estudiaremos la anatomía de la estructura de datos más importante del lenguaje: la clase. Veremos cómo declarar constantes, variables y métodos dentro de ella. También introduciremos otro tipo de dato muy importante, la estructura. Y, finalmente, explicaremos qué es el recolector de basura.

QUÉ ES UNA CLASE

Una clase es una estructura de datos que utilizaremos para definir nuestros propios tipos que extenderán los primitivos que provee el lenguaje.

La mayoría de las veces, las clases se utilizan por medio de sus instancias, las cuales se denominan objetos. Los objetos del mismo tipo (diferentes instancias de la misma clase) compartirán una estructura común, pero podrán poseer valores distintos en sus variables miembro.

Podríamos trazar una analogía entre una clase y sus objetos con lo que es un molde y los elementos creados a partir de él. Una clase define una serie de variables y comportamientos por medio de los cuales podremos crear objetos que poseerán dichas variables y comportamientos.

La cantidad de objetos que podremos crear a partir de una clase sólo estará limitada por la cantidad de memoria que posea el sistema.

CÓMO DECLARAR UNA CLASE

En C# las clases se declaran dentro de archivos de código fuente con extensión **CS**. Se utiliza esta terminación por convención, y se trata de archivos que contienen texto y que pueden de ser abiertos por cualquier editor de textos, como por ejemplo, el Bloc de notas.

A diferencia de lo que podemos ver en C++, aquí no encontraremos un archivo de declaraciones (**.H**) y otro de definiciones (**.CPP**), sino que, por el contrario, todo el código que esté relacionado con una o más clases deberá ser incluido dentro de un mismo archivo. Lo más normal es colocar una clase por archivo **CS**, pero podemos incluir la cantidad que nos parezca conveniente.

{} CLASES EN LENGUAJE JAVA

En Java, el archivo de código fuente suele terminar en ".**java**" y, a diferencia de C#, muchos compiladores sólo aceptan una clase pública por archivo.

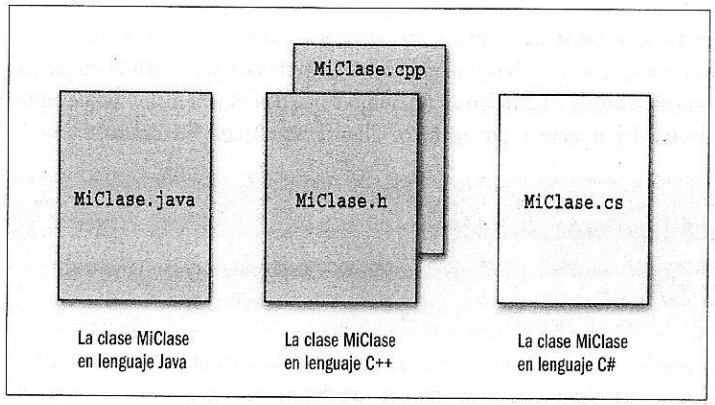


Figura 1. *Tipos de archivo utilizados para declarar clases en diversos lenguajes.*

Entonces, ¿cómo se declara una clase? Veamos:

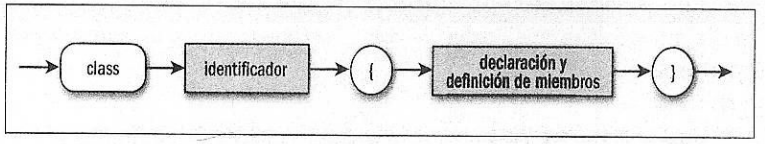


Figura 2. *Declaración de una clase.*

O en código:

```
class <identificador>
{
// cuerpo de la clase
}
```

En primer término, debemos escribir la palabra reservada **class**, y luego, un identificador válido que cumpla con las reglas que habíamos mencionado en el capítulo anterior. Este identificador debe ser único en un espacio de nombres. Algunos identificadores válidos podrían ser:

Clase1
Esta_es_una_clase
_mi_clase
miClase

Vale recordar que el hecho de que el nombre sea aceptado por el compilador no significa que éste sea un "buen" nombre. Mejor sería que el identificador de la clase fuera seleccionado en función de un nombre que describiera mejor lo que representa dentro del programa (por ejemplo, **Cliente**, **Producto**, **Cuadrado**, etc.).

Modificadores de acceso

Dentro de lo que llamamos "cuerpo" de la clase se colocarán la declaración y la definición de todos sus miembros, que serán datos y métodos.

Las variables miembro de una clase se declaran como variables convencionales (como vimos en el capítulo anterior), pero pueden poseer algunos modificadores especiales. Un modificador muy utilizado es el de **acceso**, que regula la visibilidad que posee la variable o el método desde otros objetos.

Siempre es posible acceder a las variables de una clase desde los métodos declarados en ella, más allá de su modificador de acceso. Para otros métodos de otras clases, éstos impondrán limitaciones, que son:

- **public**: si la variable es declarada como pública, entonces es posible acceder a su contenido desde cualquier objeto de cualquier tipo.
- **protected**: si la variable es declarada como protegida, entonces no es posible acceder a su contenido desde objetos externos, pero sí es posible hacerlo desde métodos de la clase y clases derivadas.
- **private**: si la variable es declarada como privada, entonces sólo es posible acceder a su contenido desde métodos de la clase, pero no desde métodos de clases derivadas. Por default, si no se especifica otra cosa, una variable es privada.
- **internal**: si la variable es declarada como interna, entonces sólo es posible acceder a su contenido desde métodos de clases que se encuentren dentro del mismo **assembly**.
- **protected internal**: si la variable es declarada como protegida interna, entonces es posible acceder a su contenido desde métodos de clases que se encuentren ubicadas dentro del mismo **assembly**, métodos que se encuentren situados en la misma clase y en clases derivadas..

C++BUILDER CON PROPIEDADES

El compilador de **Borland**, **C++Builder**, ofrece una funcionalidad de propiedades muy similar a C#. Sin embargo, esta extensión está fuera de la especificación estándar del lenguaje y requiere el uso de librerías de **Borland**, lo que restringe la portabilidad de la aplicación.

Veamos cómo declarar algunas variables:

```
class Foo
{
    int a;          // Variable del tipo entero llamada "a"
}
```

Ahora la clase **Foo** posee como uno de sus miembros a una variable del tipo número entero. Esta variable será privada, pues no se ha especificado otra cosa.

Veamos cómo incluir los modificadores de acceso:

```
class Foo
{
    public int a;
    protected int b;
    private int c;
}
```

Veamos: ahora nuestra variable **a** fue declarada del tipo pública, la variable **b** como protegida, y la **c** como privada (aunque especificar **private** no es necesario, en muchas ocasiones se realiza para remarcar su condición).

Diferencias con C++

En C++ los modificadores de acceso se especifican no por dato o método, sino por tandas de variables y métodos. En las clases, por default, sus elementos son privados, pero si escribimos **"public:"** modificamos el acceso de todo lo declarado posteriormente a público. Ejemplo:

```
class Foo
{
    int a; // variable privada
public:
    int b; // variable pública
    int c; // variable pública
```

```
    int d; // variable pública
protected:
    int e; // variable protegida
    int f; // variable protegida
private:
    int g; // Nuevamente, variable privada
}
```

Instanciación

Una vez que hemos definido nuestra clase, para poder utilizarla debemos crear una instancia de ella (aunque es posible acceder a variables y métodos estáticos sin la necesidad de crear objetos de la clase; más adelante profundizaremos en este aspecto). La declaración de un objeto es muy similar a la de cualquier otra variable. Recordemos que una clase es, al fin y al cabo, la definición de un nuevo tipo. Pues bien, **un objeto es una variable que lleva por tipo una clase**.

Suponiendo que ya hemos definido la clase **Foo**, ahora podríamos crear un objeto de ella del siguiente modo:

```
Foo miObjeto;
```

¡Pero esto no es suficiente! Si bien ya declaramos que **miObjeto** será una variable de tipo **Foo**, los objetos requieren ser definidos de manera explícita por medio del operador **new**.

```
miObjeto = new Foo();
```

Ahora sí, podremos acceder a todos los elementos del objeto. También podríamos haber declarado y definido el objeto en la misma línea:

```
Foo miObjeto = new Foo();
```

Pero ¿qué sucedería si declaráramos el objeto **miObjeto** pero no hiciéramos uso del operador **new**? El objeto poseería el valor **no definido** asociado a él y cualquier intento de acceso a sus elementos provocaría un error de compilación, ya que el compilador advertiría esta situación.

Veamos, ahora, cómo instanciar un par de objetos y acceder a variables de éstos. Supongamos que nuestra clase **Foo** posee una variable **a** y otra **b**, ambas enteras.

```
Foo obj1 = new Foo();
Foo obj2 = new Foo();
obj1.a = 10;
obj2.a = 20;
Console.WriteLine("Valor de la variable a del objeto obj1: {0}", obj1.a);
Console.WriteLine("Valor de la variable a del objeto obj2: {0}", obj2.a);
```

Tanto **obj1** como **obj2** son instancias de la clase **Foo**; sin embargo, ambos objetos son independientes entre sí. Poseen la misma estructura, pero el valor dentro de sus variables puede ser totalmente diferente.

A continuación, analicemos de qué manera hemos realizado la asignación de valores a las variables de los objetos.

No podríamos haber colocado sólo el nombre de la variable, pues el compilador no podría haber determinado a qué objeto nos estábamos refiriendo. Entonces, la sintaxis correcta es colocar el nombre del objeto, el operador de selección (el punto), la variable, el operador de asignación y, finalmente, la expresión.

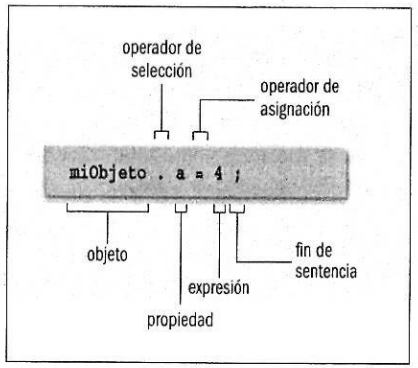


***Figura 3.** Cómo utilizar una variable miembro en la asignación.*

Constantes de una clase

Así como hablamos de constantes locales que se pueden declarar en un trozo de código, también es posible declarar constantes en el cuerpo de una clase. Las constantes pueden poseer los mismos modificadores de acceso que las variables.

```
class Foo
{
    public const int a = 10;
    // ...
}
```

Ahora **a** ya no es variable, sino constante, para todos los objetos del tipo **Foo**.

Nótese que hemos inicializado la constante en la misma línea en que la declaramos. Esto es algo obligatorio en este tipo de dato.

Pero, además, tiene una característica adicional: puede ser accedida o consultada sin requerir instanciación alguna, es decir, sin la necesidad de crear un objeto de **Foo**. ¿Por qué? Porque no tiene sentido almacenar en objetos valores que serán iguales para todos ellos sin excepción. Veamos, entonces, cómo acceder a una constante:

```
int b = Foo.a;
```

Como podemos apreciar en la línea de código anterior, creamos una variable temporal **b** y le asignamos el valor de la constante de la clase **Foo**. Nótese que no creamos un objeto de **Foo**, sino que utilizamos directamente el identificador de la clase seguido del operador selección y la constante.

Variables estáticas

Las variables estáticas, al igual que las constantes, pueden ser accedidas sin la necesidad de instanciar un objeto. Sin embargo, su valor puede ser modificado como cualquier otra variable. Podríamos pensar en ellas como **variables de clase**, siendo las variables tradicionales variables de objeto.

INICIALIZADORES

Cuando una variable se inicializa en la misma línea que su declaración dentro del cuerpo de la clase, se dice que posee un inicializador. Como vimos anteriormente, las constantes de una clase **deben** poseer un inicializador.

Este tipo de variables, al igual que las constantes o las variables convencionales, pueden poseer los mismos modificadores de acceso (**public**, **private**, **protected**, etc.).

Para declarar una variable como estática, bastará con agregarle el modificador **static** como prefijo al tipo de dato.

```
class Foo
{
    int a = 1;
    const int b = 1;
    static int c = 1;
}
```

En el listado de código anterior, hemos declarado:

- La variable **a** como un número entero. Nótese que hemos inicializado dicha variable en 1 en la misma línea que su declaración.
- La variable **b** como una constante entera.
- Y, finalmente, a **c** como una variable estática entera. También la hemos inicializado a un valor conocido, aunque esto no es obligatorio.

Veamos cómo hacer uso de este nuevo tipo de variables:

```
Foo.c = 30;              // Accedo a la variable estático
Foo obj1 = new Foo();    // Creo un objeto
obj1.a  = 10;            // Accedo a su variable a
// Muestro el valor de c
Console.WriteLine("El valor de c es: {0}", Foo.c);
// Muestro el valor de a
Console.WriteLine("El valor de obj1.a es: {0}", obj1.a);
```

Los métodos

Las clases sólo con variables y constantes no resultan demasiado interesantes. También es posible declarar funciones dentro de una clase. Estas funciones (más conocidas como métodos) contienen el código que será ejecutado por nuestro programa cuando éstas sean invocadas.
Una función miembro o método posee:

- **Un tipo de dato**: es el devuelto por la función cuando ésta finaliza.
- **Un identificador**: es el nombre por medio del cual referenciaremos al método dentro de nuestro programa; lo utilizaremos para invocarlo.
- **Parámetros**: es una lista de variables que recibe el método. Esta lista puede estar vacía o poseer tantos elementos como queramos.

En la **Figura 4** se puede apreciar un diagrama de sintaxis de la función. Cabe destacar que la función debe ser declarada dentro del cuerpo de una clase y puede poseer modificadores que veremos más adelante.

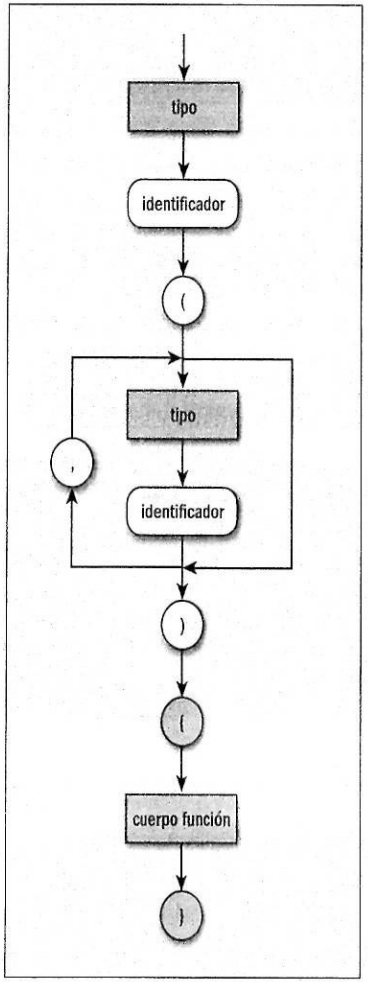


Figura 4. *Diagrama de sintaxis de una función.*

Veamos cómo declarar un método de la clase **Foo**:

```
class Foo
{
      public int valor;
      // Imprime la variable valor
      void ImprimirValor()
      {
            Console.WriteLine(valor);
      }
      // ------------------------
      // Método principal del programa
      static void Main(string[] args)
      {
            Foo obj = new Foo();
            obj.ImprimirValor();
      }
}
```

Veamos: declaramos la variable miembro **valor**. Luego, declaramos el método **ImprimirValor**, que simplemente envía a la consola de salida el valor de la variable del objeto.

Para invocar un método en un programa, siendo éste un método convencional de objeto, deberemos anteponer su nombre y el operador de selección de miembros (el punto); luego se deberá colocar entre paréntesis todos los parámetros que requiera la función (si la función no recibe parámetros, simplemente se abren y se cierran los paréntesis). Tal y como hicimos en el ejemplo anterior dentro de la función **Main**.

Métodos estáticos

Así como las variables estáticas pueden ser consideradas como variables de clase, existen métodos estáticos que pueden ser considerados como **métodos de clase**. Es-

MÁS ACERCA DEL MÉTODO MAIN

Recordemos que en nuestra aplicación, una de nuestras clases debe poseer un método estático llamado **Main**, que servirá como punto de entrada a la aplicación; es decir, que contiene el código que se ejecutará primero en nuestro programa.

tos métodos deben poseer como prefijo al tipo de dato de retorno la palabra reservada **static**. Claro que éstos tendrán algunas limitaciones: no podrán acceder a variables de objeto (es decir, variables no estáticas) o métodos de objetos (métodos convencionales no estáticos).

El modo de invocar un método estático es por medio del identificador de la clase en lugar del identificador del objeto (así como sucede con las variables estáticas).

Muchas clases de la librería BCL definen métodos estáticos. Hemos estado utilizando el método estático **WriteLine** de la clase **Console**. Ahora veamos cómo definir nuestros propios métodos estáticos.

```
class Foo
{
    // ...
    public static void ImprimirFechaActual()
    {
        Console.Write("La fecha actual es: {0}\n", DateTime.Now);
    }
    // ------------------------
    // Método principal del programa
    static void Main(string[] args)
    {
        Foo.ImprimirFechaActual();
    }
}
```

Como se puede apreciar, el método estático **ImprimirFechaActual** no hace uso de variables no estáticas del objeto, en este caso invoca un método estático de una clase de la librería BCL que se utiliza para manejar fechas.

III LA ESTRUCTURA DATETIME

El manejo de fechas solía ser un problema en muchos lenguajes. Especialmente cuando debíamos interactuar con otros programas o bases de datos, ya que el formato de almacenamiento de éstas era distinto. Pero la librería BCL ofrece una estructura llamada **DateTime** que simplifica el manejo y unifica el almacenamiento de este tipo de datos para todas las plataformas.

Luego, es conveniente notar el modo en que invocamos la función dentro de **Main**: utilizamos el nombre de la clase, no tuvimos la necesidad de instanciar ningún objeto.

Retornar valores

Hemos mencionado que los métodos pueden retornar valores o no. Si no devuelven ningún valor, simplemente se especifica que el tipo de dato de retorno sea **void**.

Veremos cómo declarar y utilizar un método cuando éste devuelve un valor entero:

```
class Foo
{
    public int valor;
    public int ValorAlCuadrado()
    {
        return valor * valor;
    }
    // ------------------------
    // Método principal del programa
    static void Main(string[] args)
    {
        Foo obj = new Foo();
        obj.valor = 9;
        Console.WriteLine("El valor del objeto obj1 al cuadrado
            es: {0}", obj.ValorAlCuadrado());
    }
}
```

Como podemos apreciar, el método **ValorAlCuadrado** retorna un tipo de dato **int**, y dentro del cuerpo de la función todos los posibles flujos de ejecución deberían terminar con alguna sentencia de retorno, caracterizada por el uso de la palabra reservada **return** y una expresión de un tipo coincidente con el valor de retorno de la función. De no existir esta sentencia de retorno dentro del cuerpo de la función, el compilador nos avisará por medio de un error en la construcción del programa. En nuestro ejemplo, se devuelve el valor multiplicado por sí mismo.

Utilizar parámetros

La declaración de parámetros en las funciones se realiza de modo muy sencillo. Básicamente, son declaraciones de variables (del modo en el que nos estamos acostumbrando a realizarlas) separadas por el carácter coma (necesario si es que hay más de un parámetro).

Analicemos el siguiente caso:

```
class Foo
{
    private int valor;
    bool FijarValor(int _val)
    {
        if (_val >= 0 && _val < 10)
        {
            valor = _val;
            return true;
        }
        else
            return false;
    }
    int DameValor()
    {
        return valor;
    }
    // ------------------------
    // Método principal del programa
    static void Main(string[] args)
    {
        Foo obj = new Foo();

        obj.FijarValor(55);
        Console.WriteLine("El valor es {0}", obj.DameValor());

        obj.FijarValor(5);
        Console.WriteLine("El valor es {0}", obj.DameValor());
    }
}
```

En el ejemplo anterior, lo que deseamos hacer es acceder a una variable privada por medio de dos métodos: uno que fija el valor a la variable y otro que la retorna. Muchas veces esta funcionalidad es deseada para realizar una verificación del tipo de dato fijado a la variable. Normalmente todas las variables miembro son privadas y se accede a ellas por medio de métodos de este tipo o a través de una funcionalidad especial que posee el lenguaje C#, que se denomina **propiedades** y que luego trataremos en detalle.

Por ahora, nos tomaremos la licencia de no utilizar propiedades y seguir con el método que deben utilizar lenguajes orientados a objetos, como C++, que no disponen de esta característica. En nuestro ejemplo, la regla que imponemos de manera arbitraria es que la variable **valor** de la clase **Foo** debe ser fijada con un rango de (0, 9). De otro modo, la función devolverá **false** y el valor no será modificado.
Analizando el cuerpo de la función **Main** podemos observar que, a pesar de que la función **FijarValor** retorna un valor booleano, éste no es utilizado para nada. Y es que nadie nos obliga a hacer uso del valor de retorno de una función. Lo negativo de nuestro accionar es que no tendremos certeza si el valor es fijado satisfactoriamente por el método, pues no nos encontramos de alertas a la información que nos brinda con respecto a esto.
Posteriormente, veamos que el valor de retorno de la función **DameValor** lo utilizamos directamente como parámetro de la función **WriteLine**; esto es algo totalmente válido y, al mismo tiempo, muy práctico.

Pasajes de valor por copia

Usualmente los parámetros se transfieren por copia de valores (a excepción del pasaje de objetos, como veremos luego). Analicemos el siguiente caso:

```
class Foo
{
    public static void IntentarModificarValor (int iValor)
    {
        iValor = 10;
    }
}
```

Si utilizamos la clase **Foo** del siguiente modo:

```
int num;
num = 99;
Foo. IntentarModificarValor(num);
Console.Write(num);
```

podremos observar que en **num**, a pesar de que la función **IntentarModificarValor** ha reescrito otra cosa, continúa conservando el valor que le hemos pasado.
Esto sucede debido a que el parámetro **iValor** de la función es una copia de la variable **num**; sólo son numéricamente iguales, pero nada más. Las dos variables se encuentran en posiciones de memoria distintas.

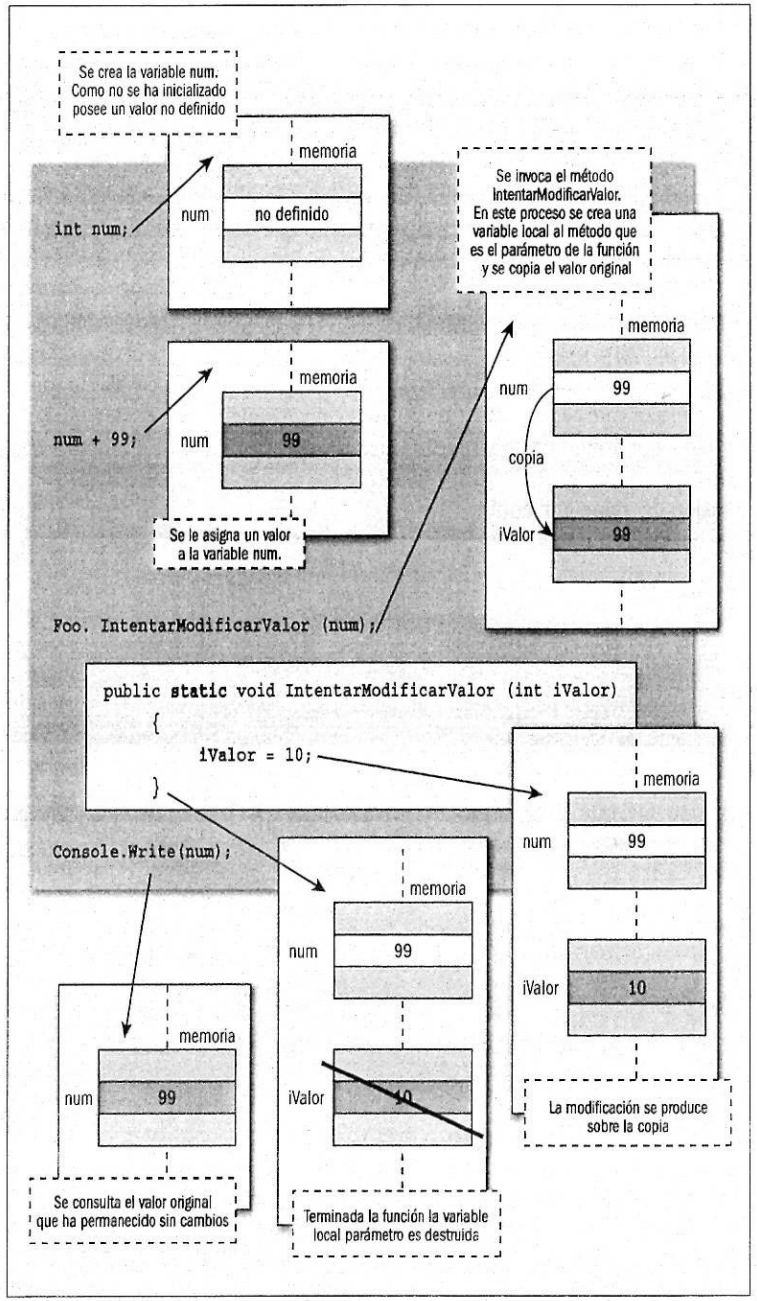


Figura 5. *Pasajes de valor por copia.*

Pasajes de valor por referencia

Pero en ciertos casos no es conveniente que el pasaje de parámetros se realice por valor. Existe un modificador, que es posible anteponer a la declaración de parámetros, que cambia el modo en que éstos son transferidos; en lugar de realizar un pasaje por copia de valores, se realiza un pasaje por referencia.

Veamos cómo realizar este tipo de pasaje de parámetros:

```
class Foo
{
    public static void IntentarModificarValor (ref int iValor)
    {
        iValor = 10;
    }
}
```

Como podemos observar en el listado anterior, el parámetro entero **iValor** posee la palabra clave **ref** antepuesta a la declaración del tipo. Luego, en el cuerpo de la función se utiliza **iValor** como si fuese una variable convencional.

```
int num;
num = 99;
Foo. IntentarModificarValor(ref num);
Console.Write(num);
```

Analizando el código anterior, podremos notar que anteponemos la palabra **ref** a la variable que pasamos como primer parámetro.

Y ahora, el método **IntentarModificarValor** realmente lo modificará. La **Figura 6** ilustra lo que ocurre paso por paso en la ejecución de este programa.

¿QUÉ ES UNA REFERENCIA?

Una referencia es básicamente una variable que contiene la dirección de memoria de otra variable pero que oculta esta naturaleza. Por lo tanto su manipulación es prácticamente idéntica a la de cualquier otra variable.

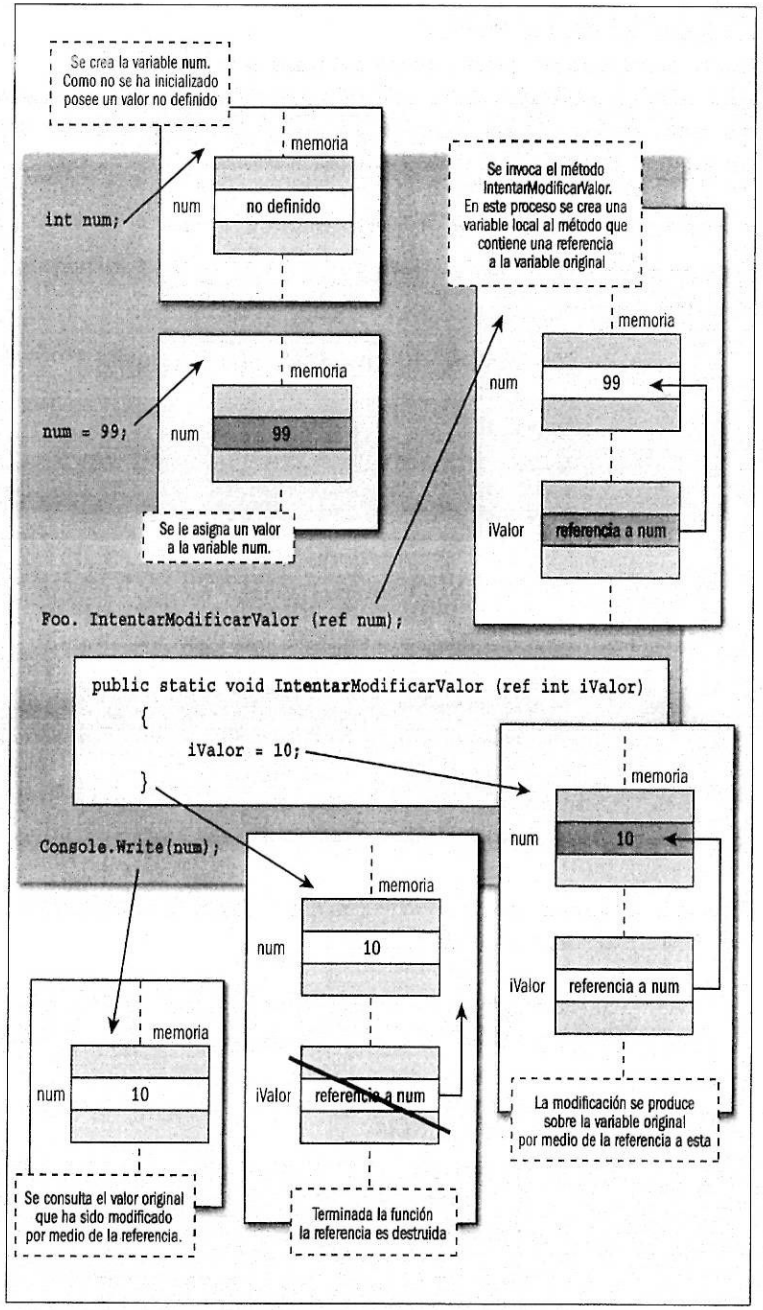


***Figura 6.** Pasaje de valor por referencia.*

Parámetros de salida

¿Qué sucede si necesitamos que nuestra función devuelva más de un valor? Existen varias alternativas. Es posible especificar que ciertos parámetros sean parámetros de salida exclusivamente por medio del modificador **out**.

Supongamos que estamos creando una clase con funciones matemáticas, algo así como la clase **Math**. Deseamos crear un método que devuelva el seno y el coseno de un número pasado como parámetro.

Siendo ésta la situación, podríamos retornar el seno como parámetro y el coseno como valor de la función, o al revés, o incluso podríamos devolver el resultado de ambas operaciones en parámetros. Veamos:

```
class MiClaseDeMatemática
{
    public static void SinCos(double dValor, out double dSin, out
        double dCos)
    {
        dSin = Math.Sin(dValor);
        dCos = Math.Cos(dValor);
    }
}
```

Como podemos observar en la lista de parámetros de la función **SinCos**, el segundo y el tercer parámetro poseen como prefijo la palabra reservada **out**. Esto significa que ambos actuarán exclusivamente como parámetros de salida.

```
double dCoseno, dSeno;
MiClaseDeMatemática.SinCos(0.78, out dCoseno, out dSeno);
Console.WriteLine("Coseno: {0}", dCoseno);
Console.WriteLine("Seno: {0}", dSeno);
```

Analicemos el código anterior:

1. Declaramos **dCoseno** y **dSeno** como dos variables locales del tipo **double**. Nótese que declaramos ambas en la misma línea, colocando el tipo de dato sólo una vez y separando los identificadores por una coma. Esto es algo totalmente válido y es un modo de tipear menos cuando debemos declarar varias variables del mis-

mo tipo. No importa que **dCoseno** y **dSeno** no sean inicializadas, la función **SinCos** deberá otorgarles un valor.

2. Invocamos el método estático **SinCos** haciendo uso del nombre de la clase, pues **SinCos** es estático.
3. Anteponemos a las variables **dCoseno** y **dSeno** la palabra reservada **out**.
4. Finalmente, imprimimos los valores retornados en parámetros.

De lo expuesto, hay dos cosas importantes que es conveniente destacar:

- La función deberá otorgar valores a todos los parámetros de salida en cualquier caso (dicho de otro modo: ningún camino de ejecución puede terminar la función sin al menos inicializar los parámetros de salida en algún valor).
- La función **SinCos** no puede leer parámetros ingresados en ella con modificador **out**: son exclusivamente parámetros de salida.

El método constructor

¿Qué ocurre si deseamos realizar cierta acción no bien el objeto es creado? ¿Deberíamos anunciarles a todos los programadores de nuestro equipo que recuerden invocar un método arbitrario luego de crear el objeto en cuestión? Por suerte, no. Existe un método especial que es invocado cada vez que un objeto nuevo es creado de forma automática.

Dicho método se denomina **constructor**, y su identificador debe ser exactamente el mismo que el de la clase de la cual forma parte. Veamos un ejemplo:

```
using System;
class Foo
{
    Foo()
    {
        Console.WriteLine("El método constructor");
    }
    // ------------------------
    // Método principal del programa
    static void Main(string[] args)
    {
        Foo f = new Foo();
    }
}
```


Si ejecutamos el programa anterior, veremos que cuando el objeto es creado, es decir, cuando realizamos un **new** de **Foo**, inmediatamente el método **Foo()** es invocado. También notemos que el método constructor al cual nos estamos refiriendo **no posee tipo de dato de retorno**. Esto es lógico, pues los constructores no se invocan como métodos convencionales, sino que son invocados automáticamente por el framework cuando se crean los objetos.

Aprovechando las características de polimorfismo que posee este lenguaje y en los que profundizaremos más adelante, es posible crear más de un constructor, cada uno con diferente cantidad de parámetros o con distintos tipos como parámetros. De este modo es posible crear un objeto al mismo tiempo que se le transfiere un parámetro.

```
class Vector2
{
    float x = 0.0f;
    float y = 0.0f;
    Vector2()
    {
        Console.WriteLine("El método constructor sin parámetros");
    }
    Vector2(float _x, float _y)
    {
        x = _x;
        y = _y;
        Console.WriteLine("El método constructor con dos
        parámetros float");
    }
    void MostrarValor()
    {
        Console.WriteLine("({0}, {1})", x, y);
    }
    // ------------------------
    // Método principal del programa
    static void Main(string[] args)
    {
        Vector2 vecA = new Vector2();
        vecA.MostrarValor();
        Vector2 vecB = new Vector2(5.0f, 10.0f);
        vecB.MostrarValor();
    }
}
```

La clase **Vector2** pretende representar vectores de dos dimensiones; para esto posee dos variables miembro (**x** e **y**) que son del tipo **float** y que son inicializadas a **0.0f** cuando el objeto es creado. Aunque las variables miembro de un objeto (¡no las variables locales!) son inicializadas a cero cuando son creadas, no está de más recordar que esto es posible. Dentro del cuerpo de la función **Main** creamos dos objetos de los dos modos posibles: invocando el constructor sin parámetros e invocando el constructor con dos parámetros.

Luego creamos un método llamado **MostrarValor**, que arroja en pantalla el contenido del objeto. Este método lo utilizamos para ver en qué valor están las variables de cada objeto inmediatamente después de construir el objeto.

El método constructor de copia

Es posible especificar un constructor de copia para poder crear un objeto que sea igual a otro en contenido de datos. Veamos cómo crear el constructor de copia de la clase **Vector2** que creamos recientemente:

```
Vector2(Vector2 copiarDesde)
{
    x = copiarDesde.x;
    y = copiarDesde.y;
    Console.WriteLine("El método constructor de copia");
}
```

Y veamos cómo utilizarlo:

```
Vector2 vecA = new Vector2();
vecA.MostrarValor();
Vector2 vecC = new Vector2(vecA);
vecC.MostrarValor();
```

III INVOCACIÓN DE CONSTRUCTORES

Es posible que un constructor invoque otro constructor. Esto suele realizarse cuando parte de la tarea de inicialización es común a todos los constructores; no tendría ningún sentido duplicar código. Simplemente se llama el constructor correspondiente -desde el otro constructor-, como si fuese un método convencional.

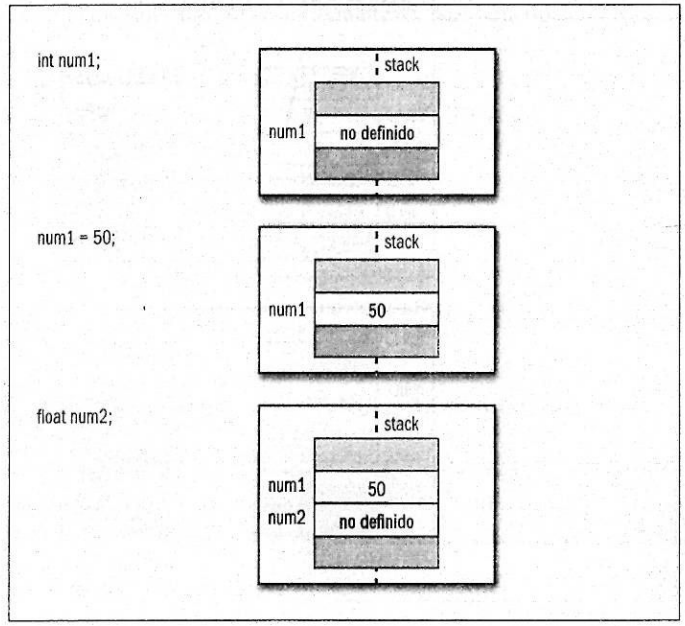
El primer vector lo creamos de la manera convencional, pero al crear el segundo enviamos el primero como parámetro. De este modo se ejecuta el constructor de copia de **vecC**, el cual toma los valores del objeto recibido y se los copia para sí. Pero ¿por qué no podríamos escribir lo siguiente?:

```
Vector2 vecA = new Vector2();
vecA.MostrarValor();
Vector2 vecC = new Vector2(vecA);
vecC = vecA; // ¡Noooooooo!
```

Para entender qué está mal con la última línea, hay que ahondar en el aspecto teórico de los objetos y cómo los maneja el lenguaje C#.

Al comienzo del **Capítulo 2** mencionábamos que existen dos tipos de datos distintos para el lenguaje: los manejados por valor y los manejados por referencia.
Los tipos de datos primitivos son manejados por valor, mientras que los objetos (instancias de clases) son manejados por referencias.

Entonces **vecA**, **vecB** y **vecC** son, en realidad, referencias. Observemos cómo funciona realmente el operador de asignación ejemplificado con variables enteras y con objetos.



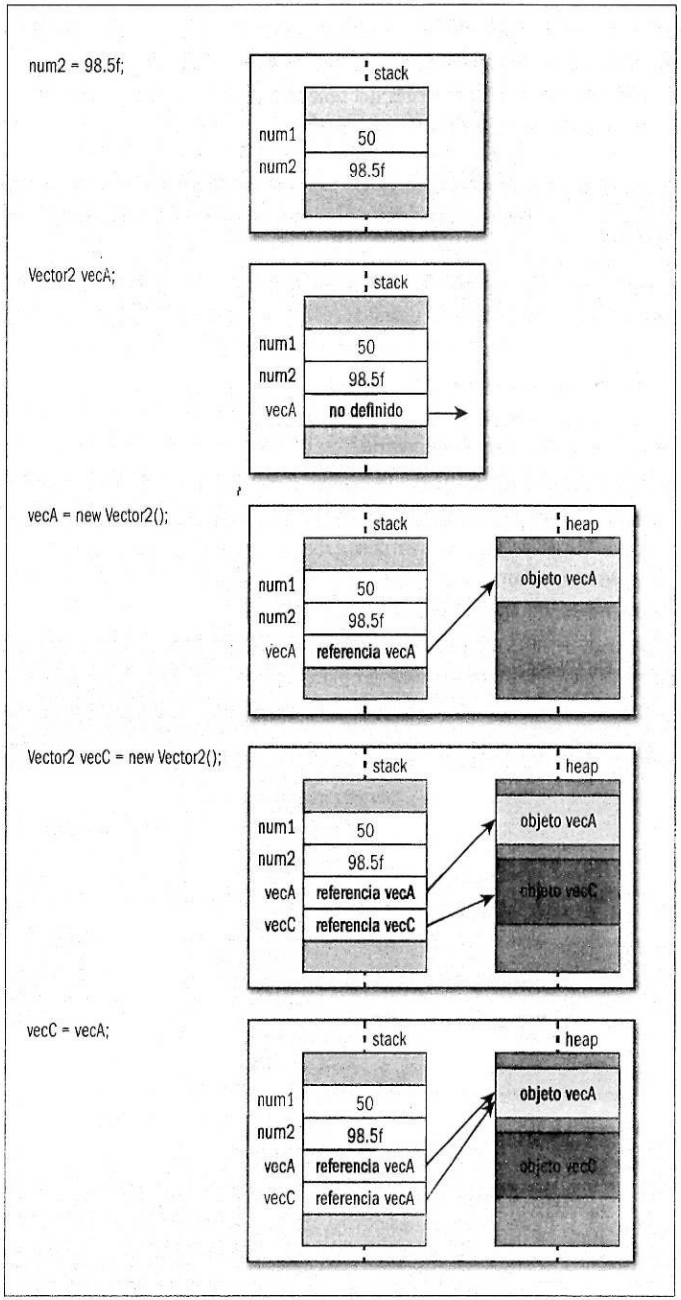


***Figura 7.** Operador de asignación.*

Detengámonos a analizar lo visto en la figura anterior. En primer lugar podemos ver cómo las variables que son de tipos de datos primitivos son creadas directamente sobre el *stack*. Aquí no existe referencia alguna, y si queremos copiar una a otra, el funcionamiento es el esperado.

Luego, podemos apreciar que los objetos, al ser manejados como referencias, poseen un comportamiento diferente. En primer lugar, el objeto en sí es creado en otra zona de memoria, llamada *heap*, y en el *stack* sólo queda una referencia a él (que es la dirección de memoria en donde se encuentra realmente).
Finalmente, si asignamos el objeto **vecA** al objeto **vecC**, lo que haremos será copiar la referencia y no el contenido del objeto. Esto podría no ser lo que esperábamos.

Por lo tanto, el modo correcto de copiar un objeto sobre otro será utilizando un constructor de copia (los programadores avanzados de otros lenguajes podrían pensar que también sería posible sobrecargar el operador de asignación, pero en C# este operador en particular no es sobrecargable).

Destrucción de objetos

Un objeto se crea cuando solicitamos memoria para él por medio del operador **new**, momento en el cual se invoca el método constructor que corresponda.

El objeto poseerá **un ciclo de vida**. Después de haber sido creado, haremos uso de él y, llegado un momento, dejará de sernos útil y desearemos poder destruirlo y recuperar la memoria que en un principio habíamos solicitado.

C# gestiona la memoria de modo automático; para esto provee un recolector de basura (**garbage collector**) que se encarga de eliminar la memoria que en algún momento se solicitó y que ya no se utiliza más (cuando, por ejemplo, se han perdido todas las referencias a ella).

En la tabla de figuras donde explicábamos cómo funciona el operador de asignación con objetos, en un determinado momento la memoria en la cual se encontraba el

III ¿CUÁNTA MEMORIA OCUPA NUESTRO PROGRAMA?

El método más sencillo -aunque no tan preciso- de averiguarlo será mirando el proceso que corresponda a nuestra aplicación desde el **Administrador de tareas**.

objeto **vecC** queda aislada, es decir que se pierde toda referencia a ella. En un lenguaje como C++ esto generaría un *memory leak*, y dicha zona de memoria quedaría reservada hasta que finalmente el programa terminara y el sistema operativo recuperara toda la memoria que estaba usando.

En cambio, C#, gracias a su recolector de basura, advertirá esta situación y recuperará dicha memoria para dejarla nuevamente disponible a nuestro programa. Pero ¿cuando ocurrirá esto?

En el lenguaje de programación C# no existe el operador contrario a **new** (como sí existe en C++, donde dicho operador se denomina **delete**). La destrucción de un objeto no es un proceso determinístico, es decir que no podremos controlar exactamente el momento en el cual la memoria asignada en el *heap* vuelve a ser memoria disponible para ser utilizada en otros objetos.

Lo que sí podemos indicarle al sistema es cuándo un objeto deja de sernos útil. Para esto bastará con pisar la referencia que teníamos de un objeto por el valor **null**:

```
// Construimos el objeto
Vector2 vecA = new Vector2();
// Hacemos uso del objeto
// ...
// Le indicamos al sistema que ya no lo precisamos
vecA = null;
```

A partir de ese momento, cuando el recolector de basura advierta esta situación, tomará cartas en el asunto. Pero el instante preciso en el que lo hará realmente queda fuera de nuestro alcance.

El recolector de basura se asegura de que:

III USO DE LA MEMORIA POR PARTE DE OBJETOS

Cada objeto instanciado ocupará un lugar en memoria, esto ya lo dijimos. Sin embargo, cabe destacar que un objeto no poseerá todos los datos en su porción de memoria: las constantes y las variables estáticas, por ejemplo, estarán ubicadas en un lugar único, relacionadas con su clase. Nótese que un objeto del tipo **Foo** poseerá una variable a totalmente independiente de las demás.

- **Los objetos sean destruidos**. En C++, un lenguaje sin gestión automática de memoria, es muy común olvidarse de destruir los objetos que ya no son utilizados, lo que genera **memory leaks**.
- **Los objetos sean destruidos sólo una vez**. Un error muy común en C++ es eliminar el mismo objeto dos veces, lo que provoca un grave error.
- **Se destruyan objetos que ya no son utilizados**. En C++ muchas veces se destruyen objetos que son referenciados aún en otras áreas del programa.

El recolector de basura trabaja cuando:

- **La memoria disponible es poca**. La búsqueda de objetos no referenciados (inalcanzables) es una tarea que consume recursos, por lo tanto sólo debe realizarse cuando es imperioso.
- **La aplicación está finalizando**.
- **El programador lo invoca manualmente**. Aunque es posible, no es recomendable (modo de invocar recolección de basura: **System.GC.Collect()**).

El destructor es un método que se invoca cuando el objeto es destruido. Allí podremos colocar el código necesario para desinicializar el objeto (cerrar archivos abiertos, conexiones de red, canales abiertos hacia dispositivos de hardware, etc.). Su declaración es similar al constructor, sólo que posee el carácter ~ como prefijo.

```
class Vector2
{
    // ...
    ~Vector2()
    {
        // Código de desinicialización
        // ...
        Console.WriteLine("Destructor");
    }
    // ...
}
```

Supongamos que tenemos un archivo abierto. Cerrar el archivo en el método destructor no estaría mal; de este modo nos aseguraríamos de que antes de que el recolector de basura eliminara nuestro objeto, el archivo estaría cerrado.

Sin embargo, el método destructor sólo puede ser invocado por el recolector de basura (y esto lo hace **justo antes** de desasignar la memoria del objeto). Como no po-

demos establecer cuándo el recolector de basura hará su trabajo, entonces tampoco podremos establecer cuándo el archivo será cerrado (o la conexión de red o el canal hacia el dispositivo de hardware), y esto sí podría ser un problema.

Una alternativa propuesta por **Microsoft** es la del patrón de diseño **Disposal**. Ésta consiste en crear un método específico que contenga todo el código de desinicialización. Este método podrá ser invocado manualmente por nosotros cuando nos plazca (a diferencia del destructor). El objeto podrá no haber sido retirado de la memoria, pero al menos las relaciones con otros elementos habrán sido debidamente cerradas. De lo que debemos asegurarnos es de que:

- Dentro de este método llamemos a **SupressFinalize**. En los siguientes párrafos veremos para qué hacemos esto.
- El método de desinicialización se cree de tal manera que pueda ser invocado en más de una oportunidad.
- Evitemos utilizar recursos desasignados por el método posteriormente.

Veamos cómo hacerlo:

```
class Vector2
{
    // ...
    public void Dispose()
    {
        // Desasigno recursos
        // ...

        Console.WriteLine("Dispose");
        GC.SupressFinalize(this);
    }
}
```

¿QUÉ SUCEDE CUANDO UN PROGRAMA AGOTA LA MEMORIA?

Cuando el sistema comienza a quedarse sin memoria, vuelca las aplicaciones que no se encuentran en uso en la memoria virtual (dentro de un gran archivo en nuestro disco). Finalmente, cuando la memoria virtual se llena, se generan excepciones y se solicita memoria al sistema.

Luego, haciendo uso de alguna instancia de **Vector2**:

```
// Construimos el objeto
Vector2 vecA = new Vector2();
// Hacemos uso del objeto
// ...
// Desasignamos recursos
vecA.Dispose();
```

Entonces, ahora invocando este método arbitrario podremos estar seguros de **cuándo** los recursos son desasignados (a excepción de la memoria ocupada por el objeto, de la cual, como mencionamos anteriormente, no poseemos control).

¿Y de qué se trata el método **SupressFinalize**? Este método recibe la referencia **this** como parámetro (que no es otra cosa que una referencia al objeto que se está ejecutando). De esta manera queremos decirle al sistema **"gracias, pero ya he desasignado todos los recursos que eran necesarios, por favor, no invoques mi método destructor"**. Cuando llegue el momento, el recolector de basura eliminará nuestro objeto del *heap* sin invocar nuestro destructor previamente, por expreso pedido nuestro.

ESTRUCTURAS

Una estructura es un tipo de dato muy similar a una clase. Es una alternativa liviana a éstas y puede contener prácticamente todos los elementos que puede poseer la clase, como variables, métodos, propiedades, constantes, etc.

La diferencia principal que posee la estructura respecto a la clase es que la estructura es un tipo de dato **manejado por valor**, a diferencia de la clase, que es manejada por referencia. Esta simple distinción repercute en muchos aspectos de uso que ya analizaremos; por ahora veamos cómo declarar una de ellas recreando la idea del **Vector2**, pero esta vez, desde una estructura:

```
struct Vector2
{
    float x;
    float y;
    public Vector3(float _x, float _y)
```

```
    {
        x = _x;
        y = _y;
    }
    void MostrarValor()
    {
        Console.WriteLine("({0}, {1})", x, y);
    }
}
```

Como podemos apreciar en el listado anterior, lo único que parece haber cambiado es la palabra clave **class** por **struct**. Sin embargo, existen otras diferencias más profundas, como:

- No es válido inicializar las variables en la misma línea que su declaración (nótese que los inicializadores están ahora ausentes).
- No es posible especificar constructores sin parámetros (nótese que hemos eliminado dicho constructor).
- No es posible especificar destructores para ellas.
- No es posible practicar herencia con ellas. Todas las estructuras son derivadas de la clase **Object** (como todo tipo de dato de C#) de manera fija, no es posible especificar que derive de ninguna otra clase, y ninguna clase puede derivar de una estructura (en el próximo capítulo ya veremos con detenimiento qué es la herencia).

Y ahora veamos diferencias en su utilización:

```
Vector3 vecA;
vecA.x = 10.0f;
vecA.y = 20.0f;
vecA.MostrarValor();
```

Analizando el código anterior, notamos inmediatamente que hemos invocado el método **MostrarValor** sin haber solicitado memoria para el "objeto". El asunto es que las variables del tipo estructura **no son objetos**. El lenguaje maneja estas variables por valor (como el resto de los tipos de datos primitivos), entonces no requieren que se solicite memoria explícitamente.

Las estructuras son asignadas en el *stack*, mientras que, por el contrario, los objetos son asignados en el *heap*.

Cabe destacar que, para poder utilizar una estructura, es necesario previamente asignarles valor a **todas** sus propiedades; de otro modo, el compilador arrojará un error.

También es posible hacer uso del operador **new** con la estructura, en especial si deseamos invocar algún constructor en particular. Todos los constructores deben proveer de valor a todas las variables de la estructura. Si no especificamos constructor, existe un constructor implícito predeterminado que inicializa todas las variables (pero es invocado sólo si utilizamos el operador **new**). Por lo tanto, también podría haber sido válido especificar:

```
Vector3 vecA = new Vector3(10.0f, 20.0f);
vecA.MostrarValor();
```

Utilizar el operador **new** no significa que la variable **vecA** sea un objeto ni que la memoria sea tomada del *heap*. Nuevamente destacamos: las estructuras son asignadas en el *stack*, **siempre**.

Por lo tanto, cuando hacemos uso de un tipo de dato de una librería, es importante saber si dicho tipo es una clase o una estructura, porque, en caso contrario, podremos cometer errores.

RESUMEN

La clase es el tipo de dato fundamental en el cual se basa el lenguaje. Realmente todo en C# es un objeto. Gracias a este recurso, crear sistemas será mucho más sencillo, y a medida que comencemos a sumar herramientas y conocimientos nuevos, veremos cómo programas que en principio parecerían complejos de realizar son muy simples y fáciles de entender. Como ya hemos mencionado, C# es uno de los lenguajes más productivos y eficientes del mercado, y gran parte de esta fama se debe a su apego a la programación orientada a objetos. En el capítulo siguiente estudiaremos los conceptos fundamentales de este paradigma y el papel que juega la estructura de datos clase en ellos.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Dónde se encuentra el error en los siguientes listados?

a)

```
class Foo
{
 const int a;
 int b;
 int c;
}
```

b)

```
class Foo
{
    public int m_iValor;
    // Imprime la variable m_iValor
    static void ImprimirValor()
    {
        Console.Write(m_iValor);
    }
}
```

c)

```
class Foo
{
    private int m_iValor;
    bool FijarValor(int iValor)
    {
      if (iValor >= 0 && iValor < 10)
      {
        m_iValor = iValor;
        return true;
      }
    }
}
```

2 ¿Es necesario instanciar una clase para acceder a:
- una variable de tipo entero no estática?
- una variable de tipo entero estática?
- un método estático?

3 ¿Cuáles son las diferencias principales entre clases y estructuras?

4 ¿Es posible especificar inicializadores en estructuras?

5 ¿Pueden las estructuras poseer constructores? ¿En qué forma?

6 ¿Es posible destruir un objeto de manera explícita?

Capítulo 4

Encapsulamiento, herencia y polimorfismo

Aquí veremos los conceptos fundamentales de la programación orientada a objetos y de qué modo C# cumple con ellos. Introduciremos conceptos primordiales, como herencia y composición. Además, aprenderemos qué es el polimorfismo y qué características de éste ofrece C#, como los métodos virtuales, las interfaces y las clases abstractas.

HERENCIA

La herencia es un concepto fundamental de la programación orientada a objetos. Por medio de esta característica podremos definir nuestras clases a partir de otras, más generales, y sólo agregar las propiedades y métodos de la especialización.
Debido a que nuestro sistema será una colección de clases relacionadas entre sí, es muy importante el modo en que estará conformada su estructura. Un diseño pobre, por lo general, genera más problemas que soluciones, ya que los errores se propagan rápidamente, y su expansión y mantenimiento se dificultan.

C#, como un buen lenguaje del paradigma de programación orientada a objetos, soporta herencia. Veamos cómo podríamos definir dos clases, una derivada de la otra, con un ejemplo minimalista:

```
class A
{
}
class B : A
{
}
```

Hemos declarado la clase **A** como una clase convencional (así como lo veníamos haciendo). La clase **B** posee, seguido de su identificador, el carácter dos puntos (:) y, luego, el identificador de la clase de la cual deriva. ¿Qué significa esto? La clase **B** heredará los métodos y las variables públicas, protegidas e internas de **A**.

¿Y de qué modo podría beneficiarnos esta característica? Bueno, si el diseño de nuestro programa es correcto, entonces escribiremos una cantidad de código menor, además de mejorar la reusabilidad de nuestros componentes y facilitar el mantenimiento de todo el sistema.

III TERMINOLOGÍA DE LA HERENCIA

Si la clase B hereda de la clase A, entonces B es una clase hija o derivada de A, mientras que A es la clase padre de B. También se suelen emplear los términos subclase (B es una subclase de A) y superclase (A es una superclase de B).

Analicemos un pequeño ejemplo. Supongamos que en nuestro sistema deberemos representar un automóvil y una pala mecánica. Ambas entidades comparten muchas características, pero también son diferentes. Un diseño pobre que no utilice herencia implementará ambas entidades en clases totalmente independientes, veamos:

Declaración de la clase Automovil:

```
class Automovil
{
    // ------------------
    // Variables
    protected int caballosDeFuerza;
    protected int cantidadPuertas;
    // ------------------
    // Métodos
    public void Arrancar()
    {
        System.Console.WriteLine("Clase Automovil, Método Arrancar");
    }
    public void Detener()
    {
        System.Console.WriteLine("Clase Automovil, Método Detener");
    }
    public void AbrirPuerta(int puertaNum)
    {
        System.Console.WriteLine("Clase Automovil, Método AbrirPuerta");
    }
    public void CerrarPuerta(int puertaNum)
    {
        System.Console.WriteLine("Clase Automovil, Método CerrarPuerta");
    }
}
```

Declaración de la clase PalaMecanica:

```
class PalaMecanica
{
    // ------------------
    // Variables
```

```
    protected int caballosDeFuerza;
    protected int pesoMaximoDeLevante;
    // ------------------
    // Métodos
    public void Arrancar()
    {
        System.Console.WriteLine("Clase PalaMecanica, Método Arrancar");
    }
    public void Detener()
    {
        System.Console.WriteLine("Clase PalaMecanica, Método Detener");
    }
    public void LevantarPala()
    {
        System.Console.WriteLine("Clase PalaMecanica,
        Método LevantarPala");
    }
    public void BajarPala()
    {
        System.Console.WriteLine("Clase PalaMecanica, Método BajarPala");
    }
}
```

Como puede verse en los listados anteriores, algunos métodos y variables son comunes a ambas clases; pero como no utilizamos herencia, tuvimos que rescribirlos.

En la **Figura 1** se puede observar un esquema de las clases descritas, que no poseen relación alguna entre sí. El diagrama de clases es uno de los diagramas **UML**, del cual se puede apreciar una introducción en el **Apéndice D**.

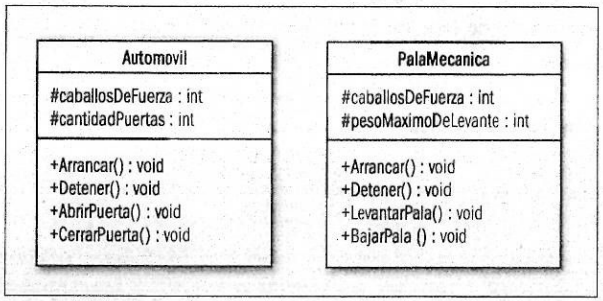


***Figura 1.** Declaración de clases sin realizar uso de herencia.*

Si, en cambio, creáramos una clase llamada **Vehiculo** que tuviese las variables y los métodos comunes a ambas clases, luego **Automovil** y **PalaMecanica** sólo deberían agregar los métodos específicos del tipo de vehículo que representan. Entonces, la cantidad de código escrito sería sensiblemente menor. También, en el caso de existir un error en el método **Arrancar**, sólo deberíamos modificar el código de una clase y no de dos. En definitiva, estaríamos ante un diseño más elegante. Veamos:

Listado de la clase Vehiculo:

```
class Vehiculo
{
    // ------------------
    // Variables
    protected int caballosDeFuerza;
    // ------------------
    // Métodos
    public void Arrancar()
    {
        System.Console.WriteLine("Clase Vehiculo, Método Arrancar");
    }
    public void Detener()
    {
        System.Console.WriteLine("Clase Vehiculo, Método Detener");
    }
}
```

Listado de la clase Automovil:

```
class Automovil : Vehiculo
{
    // ------------------
    // Variables
    protected int cantidadPuertas;
    // ------------------
    // Métodos
    public void AbrirPuerta(int puertaNum)
    {
        System.Console.WriteLine("Clase Automovil, Método AbrirPuerta");
    }
```

```
    public void CerrarPuerta(int puertaNum)
    {
        System.Console.WriteLine("Clase Automovil, Método CerrarPuerta");
    }
}
```

Listado de la clase PalaMecanica:

```
class PalaMecanica : Vehiculo
{
    // ------------------
    // Variables
    protected int pesoMaximoDeLevante;
    // ------------------
    // Métodos
    public void LevantarPala()
    {
        System.Console.WriteLine("Clase PalaMecanica,
        Método LevantarPala");
    }
    public void BajarPala()
    {
        System.Console.WriteLine("Clase PalaMecanica, Método BajarPala");
    }
}
```

Como se puede apreciar en los listados anteriores, ahora las clases **Automovil** y **PalaMecanica** son subclases de **Vehiculo**, por lo tanto todas las variables, métodos y propiedades que no sean privadas serán heredadas, y entonces podrán ser accedidas desde objetos de **Automovil** o **PalaMecanica** como si hubiesen sido declaradas allí mismo.

LA CLASE OBJECT

Decir que nuestras clases no hacen uso de herencia no es del todo preciso. Toda clase será una subclase de la clase **Object** (aunque no lo indiquemos explícitamente). Toda clase de la librería **BCL** es subclase de **Object**, que se halla en el espacio de nombres **System**. Siempre nuestros objetos podrán ser considerados del tipo **Object**, y es una ventaja que ya veremos cómo explotar.

En la **Figura 2** se ilustra la relación existente entre las clases:

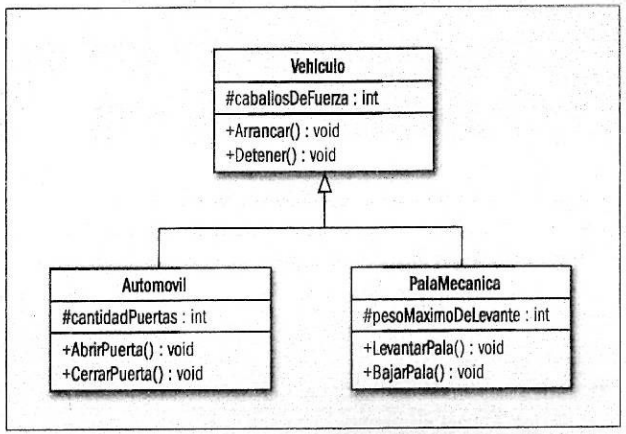


***Figura 2.** Declaración de clases haciendo uso de herencia.*

Nótese que existe una flecha grande desde las clases **Automovil** y **PalaMecanica** hacia **Vehiculo**; esta relación expresa generalización y es uno de los símbolos característicos de este tipo de diagramas.
El uso de las clases descritas no es nada nuevo; el acceso a variables o métodos propios o heredados es exactamente igual, y quien utiliza la clase en cuestión no podrá, a priori, determinar dónde fue declarado dicho elemento. Por ejemplo:

```
Automovil unAuto = new Automovil();
unAuto.Arrancar();
```

En la primera línea de código instanciamos un objeto del tipo **Automovil** y en la segunda invocamos uno de sus métodos (**Arrancar**), que realmente no fue declarado en la clase **Automovil**, sino en la clase **Vehiculo**; pero como **Automovil** es una subclase de **Vehiculo** y el método **Arrancar** es público, entonces fue heredado, y es totalmente vá-

¿DE NUEVO NEW?

Ya usamos el operador **new** para solicitar memoria al sistema. Además posee otro uso. Al definir un método (no virtual) en una subclase de una clase que contiene el mismo método, deberemos indicar que deseamos crear una versión nueva de él. No hacerlo no producirá un error, sino una advertencia. Esta regla evita que se sobrescriba un método de una clase padre por error.

lido invocarlo desde cualquier objeto del tipo **Automovil** como si hubiese estado declarado en él. Lo mismo ocurre con objetos del tipo **PalaMecanica**:

```
PalaMecanica unaPala = new PalaMecanica();
unaPala.Arrancar();
```

Ciertamente, un automóvil y una pala mecánica poseen un motor que debe ser encendido. La representación de estas entidades en nuestro sistema podría llevar a que las implementaciones de los métodos **Arrancar** y **Detener** fueran las mismas, pero tal vez podría existir el caso en que algunas de ellas o todas deberían ejecutar distintos trozos de código. En estos casos es perfectamente válido rescribir los métodos en las clases derivadas. Veamos.

La clase **Vehiculo** posee un método llamado **Arrancar**, que probablemente contenga el código necesario para el encendido de cualquier vehículo genérico. La invocación de dicho método (por medio de objetos de subclases) provocaría su ejecución. Sin embargo, si definiéramos un método **Arrancar** en la clase **Automovil** (dejando intacta la clase **Vehiculo**), entonces existirían dos métodos **Arrancar**. ¿Cuál de ellos sería el método invocado? Desde un objeto del tipo **Automovil** se invocaría el método de la clase **Automovil**, y desde un objeto del tipo **Vehiculo** se invocaría el método de la clase **Vehiculo**.

```
class Vehiculo
{
    public void Arrancar()
    {
        System.Console.WriteLine("Clase Vehiculo, Método Arrancar");
    }
}
class Automovil : Vehiculo
{
    public new void Arrancar()
    {
        System.Console.WriteLine("Clase Automovil, Método Arrancar");
    }
}
```

```
Automovil unAuto = new Automovil();
unAuto.Arrancar(); // Se invoca método de clase Automovil
```



```
Vehiculo unVehiculo = new Vehiculo();
unVehiculo.Arrancar();      // Se invoca método de clase Vehiculo
```

Esta práctica de redefinir métodos en subclases es muy común. Aunque realmente se suele invocar desde la versión del método **Arrancar** de **Automovil** al método **Arrancar** de la clase **Vehiculo**, ya que las tareas realizadas en dicha clase de otro modo quedarían ocultas y no serían ejecutadas (además de que es lógico que parte del proceso de encendido de un automóvil sea común al de cualquier vehículo). Para esto deberíamos agregar la siguiente línea de código:

```
class Automovil : Vehiculo
{
    public new void Arrancar()
    {
        base.Arrancar();
        System.Console.WriteLine("Clase Automovil, Método Arrancar");
    }
}
```

La línea nueva es:

```
base.Arrancar();
```

Base es una palabra reservada que se utiliza cuando se desea indicar al compilador que deseamos acceder a métodos, variables o propiedades de la clase padre de aquella en la que nos encontramos.

De esta manera, cuando se ejecute dicha línea de código, la versión del método **Arrancar** de la clase **Vehiculo** será ejecutada.

CUIDADO CON LA RECURSIÓN INFINITA

En el caso de que no hubiéramos utilizado **base**, y en lugar de ello hubiéramos escrito solamente **Arrancar**, entonces se habría producido un bucle infinito de ejecución del mismo método (idealmente, ya que en realidad la ejecución se detendrá en el mismo instante en que se desborde la pila y se produzca un error en tiempo de ejecución).

```
class Automovil : Vehiculo
{
    public void Arrancar()
    {
        // ¡Se invocará el mismo método una y otra vez hasta
        producir un error!
        Arrancar();
        System.Console.WriteLine("Clase Automovil, Método Arrancar");
    }
}
```

Uso de los modificadores de acceso

Las variables, por ser protegidas, no podrán ser accedidas desde fuera de la clase derivada. Es decir que la siguiente línea de código provocaría un error de compilación:

```
Automovil unAuto = new Automovil();
unAuto.cantidadPuertas = 10;
```

Así como la línea:

```
Automovil unAuto = new Automovil();
unAuto.caballosDeFuerza = 420;
```

El error en cuestión dirá que **cantidadPuertas** o **caballosDeFuerza** es inaccesible debido a su nivel de protección. Esto nada tiene que ver con la herencia, sino con los modificadores de acceso empleados en la declaración de las variables de las clases; las dos líneas de código anteriormente escritas no habrían causado ningún error si hubieran sido declaradas como públicas.

¿QUÉ ES UN ASSEMBLY?

Un sistema siempre debe estar compuesto por uno o más **assemblies**. Un **assembly** puede ser un archivo ejecutable o una librería de enlace dinámico, que puede poseer código ejecutable y/o recursos (como bitmaps, iconos o cualquier tipo de dato). Cada **assembly** posee un manifesto que describe su contenido e informa de otras propiedades como su nombre, versión, etc.

La variable **caballosDeFuerza** podrá ser accedida desde cualquier método de la clase **Vehiculo** independientemente de su modificador de acceso, por haber sido declarada dentro de esta clase. También podrá ser accedida desde los métodos de la clase **Automovil** y **PalaMecanica** debido a que fue declarada como protegida (esto no habría sido así si hubiese sido declarada como privada). El cuadro que se encuentra a continuación aclara un poco esta situación:

MODIFICADOR DE ACCESO/ ACCESIBLE DESDE	CLASE (DONDE SE DECLARÓ)	SUBCLASE (MISMO ASSEMBLY)	SUBCLASE (DISTINTO ASSEMBLY)	EXTERNAMENTE (MISMO ASSEMBLY)	EXTERNAMENTE (DISTINTO ASSEMBLY)
private	sí	no	no	no	no
internal	sí	sí	no	sí	no
protected	sí	sí	sí	no	no
protected internal	sí	sí	sí	sí	no
public	sí	sí	sí	sí	sí

***Tabla 1.** Modificadores de acceso.*

Analizando en detalle la tabla anterior, podemos inferir cuándo es conveniente utilizar cada modificador de acceso y por qué no podemos acceder externamente a las variables declaradas.
Declarar variables de clases como públicas no es una práctica aconsejable cuando:

- no todo el rango del tipo de dato de la variable debería ser aceptado;
- se deba disparar algún evento antes de la asignación o después de ella;
- la variable debe ser leída pero no escrita, o viceversa.

Por esta razón, en estas ocasiones se deberán utilizar propiedades que, a vista del "consumidor", son iguales y poseen diversos beneficios que procedemos a analizar.

Las propiedades

En lenguajes como el C++, si deseamos permitir que se modifique una propiedad de modo externo a la clase, poseemos dos alternativas:

- **Declarar la variable como pública**. Entonces, desde cualquier parte del programa se podrá leer y escribir dicha variable sin inconvenientes. El problema de esta solución es que no podremos especificar una variable como de sólo lectura o sólo escritura; si ésta es pública, siempre podrá ser leída y escrita sin excepción. Por otro lado, no podremos **reaccionar** ante la lectura o escritura de alguna variable en particular, ya sea para modificar otros valores en cascada, comprobar rangos o validez de la información especificada, etc.

- **Declarar un método de acceso a la variable**. Esta solución es la más empleada en lenguajes como C++. Por lo general se especifica un método que posee como identificador la concatenación de la palabra **Fijar** (o, en inglés, *Set*) con el identificador de la variable. En el caso de la lectura, de la variable **Leer** (o, en inglés, *Get*).

Por ejemplo, si deseamos fijar el valor de la variable **cantidad**, entonces creamos un método llamado **FijarCantidad**, y si deseamos leerla, creamos un método llamado **LeerCantidad** (claro que esto de anteponer **Fijar** y **Leer** es simplemente una convención arbitraria). Veamos:

```
class Foo
{
    protected int cantidad;
    public void FijarCantidad(int valor)
    {
        cantidad = valor;
    }
    public int LeerCantidad()
    {
        return cantidad;
    }
}
```

Luego, si es necesario, en la definición de los métodos de lectura y escritura de la variable podremos agregar revisiones de rangos y otras acciones relacionadas que puedan dispararse a partir de la lectura o la escritura de la variable en cuestión.

El único problema que posee esta solución es un empobrecimiento en la legibilidad de nuestro código, ya que es más bonito especificar:

```
objeto.variable = valor;
```

que

```
objeto.método(valor);
```

Sobre todo, cuando debemos acceder a variables de objetos que se encuentran dentro de otros objetos, con lo que en el primer caso quedará:


```
objeto.objeto.objeto.variable = valor;
```

y en el segundo caso:

```
objeto.método().método().método(valor);
```

Las **propiedades**, ausentes en C++, pero presentes en lenguajes de programación más modernos como C#, nos brindan la posibilidad de tomar lo mejor de los dos mundos: la posibilidad de disparar código relacionado con la lectura o escritura de una variable sin perder el estilo de codificación.

```
class Foo
{
    protected int cantidad;
    public int Cantidad
    {
        get
        {
            return cantidad;
        }
        set
        {
            cantidad = value;
        }
    }
}
```

Analicemos el código anterior. Por un lado nos encontramos con la declaración de la variable (como protegida, en este caso, para que pueda ser accedida desde subclases, pero no externamente). Por el otro, existe la declaración de una **propiedad** que posee un modificador de acceso y un tipo de dato como si fuese una variable o un método convencional, pero inmediatamente después de su identificador posee un bloque declarativo dentro del cual existe un **get** (*leer*) y un **set** (*fijar*), que, a su vez, poseen bloques donde colocaremos código.

El hecho de que el identificador utilizado sea similar a la variable a la cual accede (sólo cambia que la primera letra está en mayúscula) es una convención arbitraria nuestra; podríamos haber especificado un identificador totalmente diferente.

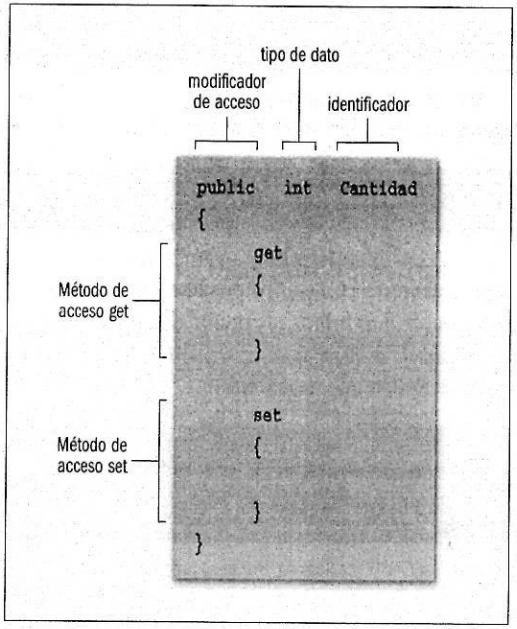


Figura 3. *Ejemplo de declaración de una propiedad.*

Ya podemos intuir que el código que colocaremos dentro del **get** será el que antes especificamos dentro del **LeerCantidad**. Así como el código que colocaremos dentro del **set** será el que antes especificamos dentro del **FijarCantidad** (en este último caso, debido a que no existe declaración de parámetro alguno, deberemos utilizar la palabra reservada **value**, que oficiará de parámetro de entrada).

Entonces, si deseamos invocar el método **get**, podremos escribir:

```
int valor = unObjetoFoo.Cantidad;
```

AL MENOS UNO A LA VEZ

Cabe destacar que podremos especificar el **get** y el **set** o sólo uno de ellos (si es que deseamos que nuestra variable sea de sólo lectura o de sólo escritura), pero al menos uno debe estar declarado, o se producirá un error de compilación.

Y si deseamos invocar el método **set**:

```
unObjetoFoo.Cantidad = 10;
```

Es decir, mantenemos la simpleza de lectura y escritura de variables, pero con el poder que nos otorga la ejecución de código arbitrario dentro de estos métodos especiales.

El siguiente diagrama expone cómo debe declararse una propiedad. Cabe destacar que ésta podría poseer más modificadores (como **static** si la variable fuese estática), que más adelante analizaremos.

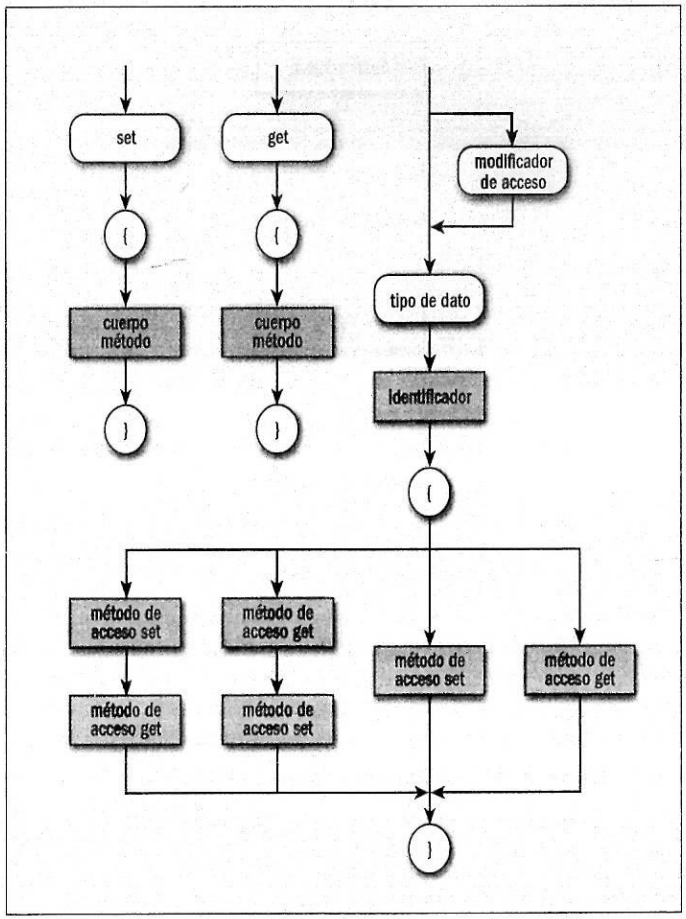


Figura 4. *Declaración de una propiedad.*

INVOCAR CONSTRUCTORES

Sabemos que cuando construimos un nuevo objeto se ejecuta el código del método constructor. Dicho constructor puede existir de manera explícita (cuando lo definimos) o de manera implícita (cuando no lo definimos).
Pero ¿qué es lo que sucede cuando nuestra clase está definida a partir de otra? ¿Qué constructor se invoca? Para responder estas preguntas, pasemos a analizar con detenimiento el siguiente ejemplo:

Poseemos tres clases: **A**, **B** y **C**, cada una de ellas definida a partir de la otra, según detalla el siguiente diagrama:

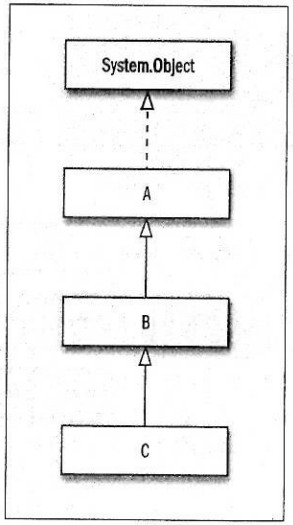


Figura 5. *Nuestro diagrama de clases.*

Si creamos una instancia de la clase **C**, entonces, en primer término se pasará a ejecutar el constructor de la clase padre (que, en realidad, siempre tendría que ser **Object**). Luego se ejecutará el constructor de la clase **A**, después el constructor de la clase **B** y, finalmente, el constructor de la clase **C**.

Esto podremos verificarlo con el siguiente ejemplo:

```
class A
{
    A()
```



```
    {
        Console.WriteLine("Constructor A");
    }
}

class B : A
{
    B()
    {
        Console.WriteLine("Constructor B");
    }
}
class C : B
{
    C()
    {
        Console.WriteLine("Constructor C");
    }
    static void Main(string[] args)
    {
        C obj = new C();
    }
}
```

La salida en pantalla será:

```
Constructor A
Constructor B
Constructor C
```

Esto que observamos es lógico, debido a que el objeto **C**, en el momento de ser construido, debería tener la posibilidad de contar con todas las variables y propiedades de sus superclases debidamente inicializadas (esto, en un gran número de ocasiones, podría significar abrir archivos o canales hacia otros dispositivos de hardware), y así realizar su propia inicialización.

¿Y qué ocurriría si deseáramos invocar un constructor con parámetro en alguna clase padre? Veamos el siguiente ejemplo:

```
class B : A
{
      B()
      {
            Console.WriteLine("Constructor B");
      }
       B(int num)
      {
            Console.WriteLine("Constructor B con parámetro
            entero: {0}", num);
      }
}
```

Ahora nuestra clase **B** cuenta con dos constructores; uno de ellos recibe un parámetro que es una variable del tipo entero.

Pero en el caso de que volvamos a ejecutar nuestro programa, el constructor de la clase **C** seguirá ejecutando el constructor sin parámetros de **B**. ¿Cómo hacemos para cambiar este comportamiento?

```
class C : B
{
      C() : base(1)
      {
            Console.WriteLine("Constructor C");
      }

      // ...
}
```

¡Haciendo uso de la palabra reservada **base**! Tengamos especial atención en que hemos colocado un carácter dos puntos (**:**) tras cerrar los paréntesis en la declaración del constructor de la clase **C**, luego escribimos **base** (que hace referencia a la clase padre inmediata desde la cual estamos), y entre paréntesis especificamos los parámetros que hagan falta.

Este parámetro, en el ejemplo anterior, fue un número arbitrario, pero también podría ser una variable recibida por parámetro en un constructor de la clase **C** (**Figura 6**).


```
class A
{
      public A()
      {
            Console.WriteLine("Constructor A");
      }
}
class B : A
{
      public B()
      {
            Console.WriteLine("Constructor B");
      }
      public B(int num)
            Console.WriteLine("Constructor B con parametro
                  entero: {0}", num);
      }
}
class C : B
{
      public C()
      {
            Console.WriteLine("Constructor C");
      }
      public C(int num) : base(num)
      {
            Console.WriteLine("Constructor C con parametro
                  entero: {0}", num);
      }
      static void Main(string[] args)
      {
            c obj = new c(5);
      }
}
```

Figura 6. *El parámetro del constructor de C es pasado a B.*

También cambiamos el modo de construir un objeto del tipo **C**, para que se invoque el constructor con parámetro entero y no el constructor sin parámetros.

INVOCAR DESTRUCTORES

En el momento en que un objeto es destruido, los destructores también son invocados en un orden preestablecido, que, como es de esperar, es inverso al ordenamiento que siguen los constructores.

```
class A
{
    A()
    {
        Console.WriteLine("Constructor A");
    }

    ~A()
    {
        Console.WriteLine("Destructor A");
    }
}
class B : A
{
    B()
    {
        Console.WriteLine("Constructor B");
    }
    ~B()
    {
        Console.WriteLine("Destructor B");
    }
}
class C : B
{
```

DESTRUCCIÓN DE OBJETOS

Recordemos que la destrucción real de un objeto es realizada por el framework, y deberíamos dejar esta tarea en sus manos y no presuponer cuándo un objeto es destruido.


```
    C()
    {
        Console.WriteLine("Constructor C");
    }
    ~C()
    {
        Console.WriteLine("Destructor C");
    }
    static void Main(string[] args)
    {
        C obj = new C();
    }
}
```

Si ejecutamos el ejemplo anterior, veremos en pantalla:

```
Constructor A
Constructor B
Constructor C
Destructor C
Destructor B
Destructor A
```

Antes de ser destruido por el framework, el objeto **C** invoca su método destructor y le permite a éste realizar un proceso inverso a la inicialización (cerrar archivos, canales, etc.). Luego, se invoca el destructor de la clase padre de **C**, y el proceso se repite hasta llegar a la clase raíz, que siempre será **System.Object**.

COMPOSICIÓN

Una clase puede tener como miembro cualquier tipo de variable, sin importar que sea de algún tipo fundamental o definido por el usuario; esto también incluye otros objetos definidos previamente.
Por tal motivo, en un sistema será bastante frecuente que los objetos se encuentren compuestos por otros objetos. Veamos un ejemplo trivial.

```
class A
{
    public void m1()
    {
        Console.WriteLine("Clase A, método Foo");
    }
    // ...
}
class B
{
    public A objA = new A();
    // ...
}
```

En el ejemplo mostrado anteriormente, todo objeto de la clase **B** se encontrará conformado por un objeto de la clase **A**.

Veamos cómo representar esta nueva relación en un diagrama de clases:

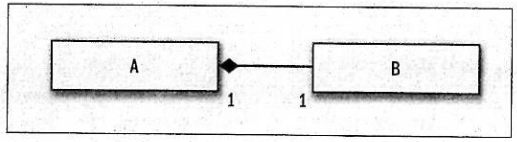


***Figura 6.** La relación de composición.*

Como se puede apreciar en el diagrama, la relación de composición se marca por medio de una línea terminada en diamante lleno.

A través de la utilización de un objeto de la clase **B**, podremos tener acceso al objeto **A** por medio de la variable **objA** (que hicimos pública) y, de esta manera, acceder luego poder a sus variables, propiedades y métodos públicos.

```
B unObjB = new B();
// Invoco el método m1 por medio del objeto objA del tipo A
unObjB.objA.m1();
```

De este modo, si el objeto del tipo **A** contuviera, a su vez, otros objetos, podríamos acceder a ellos simplemente anteponiendo el operador de selección de miembros, siempre y cuando éstos fueran públicos.

COMPOSICIÓN FRENTE A HERENCIA

La composición y la herencia tienen bastante en común, y en muchas ocasiones resulta difícil discernir cuál de ellas utilizar en un caso determinado. Veamos cuáles son sus similitudes. Si tenemos la siguiente clase:

```
class A
{
    public int var1;
    public int var2;
    public int var3;
}
```

en memoria, al crear un objeto del tipo **A** tendremos lo siguiente:

objA	var1
	var2
	var3

Ahora, si creamos una clase **B**, subclase de **A**, y agregamos un par de variables, tendremos lo siguiente:

```
class B : A
{
    public int var4;
    public int var5;
}
```

Cuando instanciemos un objeto del tipo **B**, el mapa de memoria para éste será:

objB	var1
	var2
	var3
	var4
	var5

Es decir que el objeto del tipo **B** poseerá las variables de todas las clases superiores (en este caso, sólo una, sin contar la clase **Object**, que, de todos modos, no posee variables) más las definidas por él.

Por lo tanto, un objeto del tipo **B** podrá acceder a elementos de **B** y a elementos (heredados) de **A**. Podríamos haber obtenido un resultado similar utilizando composición. Veamos:

La clase **A** permanecería sin cambio alguno. Sin embargo, la clase **B** ya no sería una subclase de **A**, sino que poseería otra variable más, que sería un objeto del tipo **A**:

```
class B
{
    public A objA = new A();
    public int var4;
    public int var5;
}
```

objB	objA	var1
		var2
		var3
	var4	
	var5	

El mapa de memoria es muy similar; el objeto, en definitiva, ocupará el mismo espacio en memoria. ¿Cuál es la diferencia, entonces?

Usualmente elegimos utilizar herencia cuando deseamos incorporar a nuestra clase las variables, propiedades y métodos del objeto seleccionado, o cuando deseamos especializar una clase determinada agregando características de una entidad específica que estamos modelando, tomando como base una clase del mismo tipo, pero más general.
En cambio, decidimos utilizar la composición cuando deseamos ocultar, encapsular, el objeto seleccionado bajo la nueva interfaz.
Por lo tanto, más allá de algunas similitudes técnicas, conceptualmente la herencia y la composición se encuentran bien diferenciadas.

INICIALIZAR OBJETOS MIEMBRO

Cabe destacar que no es estrictamente necesaria la creación del objeto **objA** en la misma línea que su declaración (inicializador); esta operación podría realizarse luego. Sin embargo, es una práctica conveniente en muchos casos, ya que evitará que por error se haga uso del objeto **objA** sin antes haberlo creado.

POLIMORFISMO

El polimorfismo es la habilidad que poseen los objetos para reaccionar de modo diferente ante los mismos mensajes. Un objeto del tipo **Puerta**, al igual que un objeto del tipo **Ventana**, podrá recibir el mensaje **Abrir**; sin embargo, cada uno de ellos reaccionará de modo diferente.

En C#, el polimorfismo se encuentra íntimamente relacionado con el mecanismo de sobrecarga y con los métodos virtuales. Veamos.

Sobrecarga de métodos

En lenguajes como el C, es común encontrar librerías que poseen grupos de funciones que realizan la misma tarea, pero con distintos tipos de datos en los parámetros. De esta manera, si la tarea consiste en mostrar en pantalla el dato pasado como parámetro, nos podremos encontrar con casos como el siguiente:

```
void ImprimirInt(int num);
void ImprimirChar(char car);
void ImprimirFloat(float num);
```

El lenguaje C# provee un mecanismo mucho más conveniente para obtener el mismo propósito. Es posible crear métodos de clases que posean el mismo nombre y difieran en la cantidad y/o en el tipo de parámetros. Esto no es algo realmente novedoso para nosotros, pues ya hemos observado el mecanismo en acción definiendo múltiples constructores.
De este modo, el ejemplo anterior podría ser rescrito como:

```
class Con
{
    static void Imprimir(int num);
    static void Imprimir(char car);
    static void Imprimir(float num);
}
```

Lo cual es mucho más sensato. De esta forma, el programador no tendrá que recordar el sufijo que conforma el identificador de la función apropiada. Todas las funciones se llaman de la misma manera. A esta característica del lenguaje se la ha denominado **sobrecarga** de métodos.

Cuando el programador utilice alguna de estas funciones, el compilador seleccionará el método adecuado en función de los tipos de datos y la cantidad de parámetros que utilicemos, invocando la versión del método que corresponda.

Tipo de dato devuelto por el método

No es posible crear dos funciones con el mismo identificador que sólo difieran en el tipo de dato devuelto, debido a que el compilador no podrá distinguir cuándo se desea llamar a una o a otra:

```
class Foo
{
    // Error: Los métodos sólo difieren en el tipo
    // de datos retornado.static bool Foo(int num);
    static bool Foo(int num);
    static char Foo(int num);

}
```

Veamos un ejemplo un poco más interesante. Crearemos una clase que manipule un vector de dos dimensiones, completando la clase que habíamos empezado a construir en el **Capítulo 3**. Empezaremos donde antes habíamos terminado. Una clase con dos variables de tipo **float** privadas que representan los dos componentes de nuestro vector. Ahora, la definición de dos constructores: uno sin parámetros y otro con parámetros, para poder fijar el valor del vector al momento de construirlo, lo cual es muy cómodo.

```
class Vector2
{
    public float X = 0.0f;
    public float Y = 0.0f;

    Vector2()
    {
    }
    Vector2(float _x, float _y)
    {
        X = _x;
        Y = _y;
    }
}
```


Debido a que todo el rango de flotantes es aceptado por nuestro vector para sus componentes y a que no debemos disparar ninguna acción relacionada con la lectura o escritura de éstos, entonces las variables miembro las especificaremos con modificador de acceso **public**.
Agregaremos ahora algunos métodos útiles para la manipulación de vectores:

```
public double ProductoPunto(Vector2 vec2)
{
    return X*vec2.X + Y*vec2.Y;
}
public double ProductoPunto(float _x, float _y)
{
    return X*_x + Y*_y;
}
```

El método **ProductoPunto** devuelve el producto punto del vector almacenado en el objeto y el pasado como parámetro.
El método se encuentra sobrecargado y permite pasar el vector con el cual realizar el producto punto como un objeto del tipo **Vector2** o mediante los elementos de éste en tipo de dato **float**.
De la misma manera, podríamos definir un método que nos permitiera sumar y restar dos vectores. Veamos:

```
public void Sumar(Vector2 vec2)
{
    X = X + vec2.X;
    Y = Y + vec2.Y;
}
public void Sumar(float _x, float _y)
{
```

III PARÁMETROS OPCIONALES

C# no soporta la declaración de parámetros opcionales, así como lo hacen C++ o Visual Basic. Para conseguir el mismo objetivo, habrá que crear dos métodos distintos sobrecargados con diferencia en cantidad de parámetros.

```
        X = X + _x;
        Y = Y + _y;
    }
    static public Vector2 Sumar(Vector2 vec1, Vector2 vec2)
    {
        return new Vector2(vec1.X + vec2.X, vec1.Y + vec2.Y);
    }
```

La primera versión del método suma un vector pasado como parámetro al vector del objeto y almacena el resultado sobre sí mismo. La segunda versión hace lo mismo, pero recibe los parámetros como elementos.

En último lugar, la tercera versión del método es estática y no requiere la existencia de un objeto, simplemente suma dos vectores y devuelve el resultado de la operación como valor de la función.

Métodos estáticos y no estáticos en la sobrecarga

No es posible que dos métodos sobrecargados sólo difieran en el modificador **static**. Por ejemplo:

```
public void Foo (int num)
{
// ...
}
static public void Foo (int num)
{
// ...
}
```

El compilador tomará el código anterior como un error.

ORDEN DE LOS PARÁMETROS

Los métodos sobrecargados tampoco podrán diferenciarse exclusivamente en el orden de los parámetros (si es que éstos son del mismo tipo), ya que el compilador no podrá diferenciar en qué caso invocar qué método.

Sobrecarga de operadores

Sería mucho más práctico poder hacer la suma de dos vectores utilizando el operador de suma (+), en particular la última versión del método **Sumar**. De otro modo, así como estamos ahora, deberíamos escribir:

```
vec3 = Vector2.Sumar(vec1, vec2);
```

Cuando sería más elegante poder escribir:

```
vec3 = vec1 + vec2;
```

Pero ¿cómo podría saber el lenguaje cómo debe manipular dos vectores de dos dimensiones en una suma? O, eventualmente, ¿cómo podría saber cómo "sumar" dos objetos definidos arbitrariamente por nosotros? C# permite **sobrecargar** operadores, es decir, **redefinir el significado** de un conjunto de operadores cuando se utilizan con ciertos objetos.
Esta sobrecarga devendrá en un método convencional pero declarado de un modo especial, por lo que termina siendo, al fin y al cabo, una modificación a la estética de nuestro código.
Veamos cómo sobrecargar el operador +:

```
static public Vector2 operator+ (Vector2 vec1, Vector2 vec2)
{
    return new Vector2(vec1.X + vec2.X, vec1.Y + vec2.Y);
}
```

Como se puede apreciar en el listado anterior, la declaración del operador es muy similar a la de un método convencional, sólo que el identificador de éste se encuentra formado por la palabra **operator** y el operador que se desea sobrecargar.

OPERADORES SOBRECARGABLES

¡No todos los operadores pueden ser sobrecargados! En este mismo capítulo podremos encontrar una lista de los operadores que podremos sobrecargar en el lenguaje C#.

El método que define la sobrecarga de un operador **debe ser público y estático.**

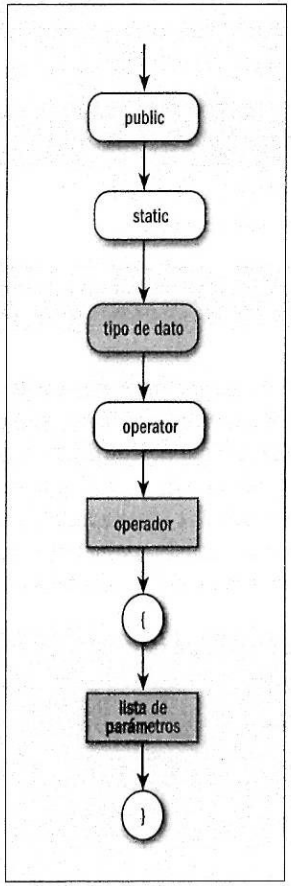


Figura 7. *Sobrecarga de un operador.*

Ahora, el siguiente código es válido:

```
Vector2 vec3 = vec1 + vec2;
```

Sin embargo, de todos los operadores con los que cuenta C#, no todos son sobrecargables. En la siguiente tabla podemos apreciar qué operadores pueden ser sobrecargados y el tipo al cual pertenecen:

TIPO DE OPERADOR	OPERADORES	DESCRIPCIÓN
Unarios	+	Indica valor positivo
	-	Negación aritmética
	!	Negación lógica
	~	Negación bitwise
	&	Y bitwise
	|	O bitwise
	++	Incremento
	--	Decremento
	true	Verdadero
	false	Falso
Binarios	+	Suma
	-	Resta
	*	Multiplicación
	/	División
	%	Resto
	&	Y bitwise
	|	O bitwise
	^	O exclusivo bitwise
	<<	Desplazamiento a izquierda
	>>	Desplazamiento a derecha
Relacionales	==	Igualdad
	!=	Desigualdad
	<	Menor
	>	Mayor
	<=	Menor o igual
	>=	Mayor o igual

Tabla 2. *Tabla de operadores sobrecargables.*

Sobrecarga de operadores unarios

Veamos el siguiente ejemplo:

```
class Vector2
{
    // ...
    static public Vector2 operator- (Vector2 vec1, Vector2 vec2)
    {
        return new Vector2(vec1.X - vec2.X, vec1.Y - vec2.Y);
    }
}
```

Sobrecarga de operadores binarios

En el siguiente ejemplo sobrecargaremos los operadores **true** y **false** (si se sobrecarga uno de ellos, en compilador nos obliga a sobrecargar los dos). Este operador se invoca cuando se evalúa una expresión con el objeto en cuestión:

```
class Vector2
{
    // ...
    static public bool operator true(Vector2 vec)
    {
        return (vec.X != 0 && vec.Y != 0);
    }
    static public bool operator false(Vector2 vec)
    {
        return !(vec.X != 0 && vec.Y != 0);
    }
}
```

Sobrecarga de operadores relacionales

En el siguiente ejemplo, especificaremos cuándo dos vectores deben ser considerados iguales y cuándo, distintos (al igual que con los operadores **true** y **false**, aquí también el operador == y el != deben ser sobrecargados en pares).

```
class Vector2
{
    // ...
    static public bool operator == (Vector2 vec1, Vector2 vec2)
    {
        return (vec1.X == vec2.X && vec1.Y == vec2.Y);
    }
    static public bool operator != (Vector2 vec1, Vector2 vec2)
    {
        return (vec1.X != vec2.X || vec1.Y != vec2.Y);
    }
}
```

Los siguientes operadores no son sobrecargables:

TIPO DE OPERADOR	OPERADORES	DESCRIPCIÓN
Primario	()	Cast
	.	Selección
	new	Asignación de memoria
	typeof	Tipo
	sizeof	Tamaño
	checked	Verificado
	unchecked	No verificado
Condicional	&&	Comparación Y
	||	Comparación O
	?:	Condicional
Asignación	=	Asignación
	+=	Suma y asignación
	-=	Resta y asignación
	*=	Multiplicación y asignación
	/=	División y asignación
	%=	Resto y asignación
	&=	Y bitwise y asignación
	^=	O exclusivo y asignación
	<<=	Desplazamiento a izquierda y asignación
	>>=	Desplazamiento a derecha y asignación
	|	O bitwise
Relacional	is	Operador de verificación de tipo
Selección	->	Selección de miembro
Indexamiento de array	[]	Indexador

***Tabla 3.** Tabla de operadores no sobrecargables.*

Métodos virtuales

Una de las características más importantes que ofrece el lenguaje en cuanto a polimorfismo tiene que ver con los métodos virtuales.

III INDEXADORES

El operador [] (indexación de array) no puede ser sobrecargado directamente. Sin embargo, C# dispone de un recurso denominado indexadores, que nos permite realizar una tarea equivalente que ya veremos en el **Capítulo 5**.

La motivación principal de la creación de este tipo de métodos es la de poder separar la interfaz de una entidad con su implementación específica, lo que nos permite aumentar en gran medida la reusabilidad de nuestro código y crear diseños mucho más elegantes. Veamos, por medio de un ejemplo, cuál es el beneficio de este tipo de métodos.

Supongamos que estamos programando un juego de ajedrez. En dicho juego deseamos crear distintas **inteligencias** que estén codificadas de modo diferente, pero que posean la misma interfaz, ya que, al fin y al cabo, están jugando al mismo juego.

Tendremos una clase **Jugador**, donde deberíamos implementar una cierta lógica de juego. Tengamos en cuenta que deseamos implementar varias estrategias. En cuanto a métodos, la clase poseerá sólo uno, que será **RealizarMovida**, donde entrará en acción la lógica para evaluar el juego y realizar una movida.

La declaración de la clase podría ser la siguiente:

```
class Jugador
{
    public void RealizarMovida()
    {
        Console.WriteLine("Clase: Jugador, Método: RealizarMovida");
    }
}
```

Luego tendremos la clase **Juego**, que poseerá dos variables y un método:

Jugador jugador1: objeto que representará el primer jugador.
Jugador jugador2: objeto que representará el segundo jugador.
EmpezarPartida: método que comenzará la partida. En nuestro ejemplo minimalista, simplemente invocará el método **RealizarMovida** de **jugador1** y de **jugador2**.

Declaración de la clase **Juego**:

```
class Juego
{
    Jugador jugador1;
    Jugador jugador2;
    public void EmpezarPartida(Jugador jug1, Jugador jug2)
```



```
    {
        jugador1 = jug1;
        jugador2 = jug2;
        jugador1.RealizarMovida();
        jugador2.RealizarMovida();
    }
}
```

Nuestro modelo de clases terminará allí, debido a que simplemente queremos mostrar una funcionalidad específica.
No existirá tablero ni fichas, sino que, por el contrario, sólo mostraremos en pantalla algunos mensajes para analizar de qué manera se comporta el compilador al ser sometido ante distintos modelos.
Nuestro sencillo diagrama estático de clases podría ser:

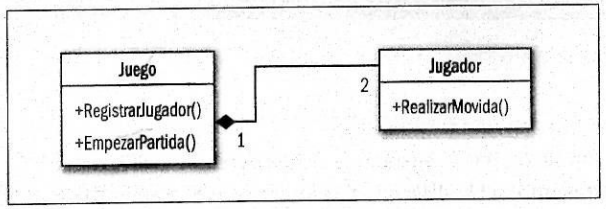


***Figura 8.** Diagrama de clases.*

Ahora bien, ¿cómo haremos uso de nuestras clases? En un primer ejemplo, lo haremos instanciando la clase **Jugador** en dos ocasiones:

```
class Foo
{
    static void Main(string[] args)
    {
        Jugador jug1 = new Bot1();
        Jugador jug2 = new Bot2();
        Juego j = new Juego();
        j.EmpezarPartida(jug1, jug2);
    }
}
```

Ejecutar nuestro programa en este momento producirá en pantalla la siguiente salida:

```
Console.WriteLine("Clase: Jugador, Método: RealizarMovida");
Console.WriteLine("Clase: Jugador, Método: RealizarMovida");
```

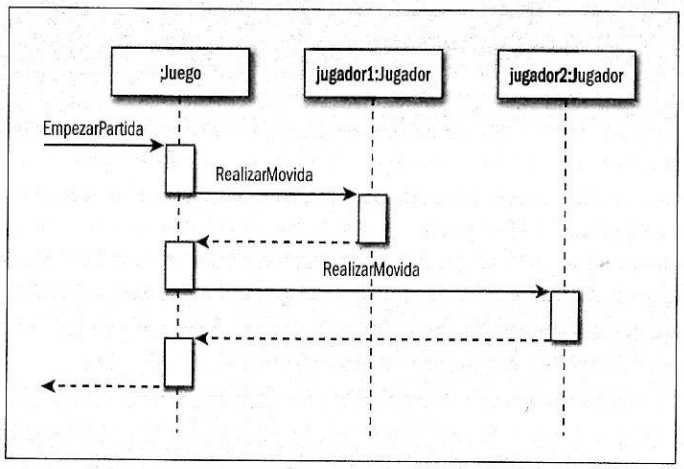


***Figura 9.** Diagrama de secuencias.*

El diagrama de secuencias anterior nos permite ver en forma gráfica cómo se deben realizar las llamadas. Un diagrama de secuencias se encuentra compuesto por distintos elementos:

- **Objetos**: son representados por los rectángulos no llenos en el borde superior del diagrama. Dentro de ellos se debe colocar su nombre seguido de unos dos puntos y el nombre de la clase. En caso de querer indicar que lo especificado es para cualquier objeto del tipo, se colocan simplemente unos dos puntos y el nombre de la clase.
- **Mensajes**: son las líneas horizontales que cruzan el diagrama. Representan invocaciones a funciones o métodos, y las hay de distintos tipos (simple, sincrónica y asincrónica).

III EL MÉTODO TOSTRING DE LA CLASE OBJECT

Existe un método virtual en la clase **Object** que normalmente sobrecargaremos en nuestras clases. Dicho método es ejecutado por el framework cuando se desea obtener una representación en string de un objeto determinado. Si no sobrecargamos el método, el comportamiento predeterminado será mostrar el nombre de nuestra clase en pantalla.

- **Tiempo**: las líneas verticales representan el tiempo. Es decir que lo que se coloque más arriba se producirá antes que lo que se coloque más abajo. Los rectángulos colocados sobre estas líneas de tiempo representan un flujo de ejecución determinado.

En el **Apéndice D** se puede encontrar más información respecto a este diagrama y los otros diagramas del lenguaje de modelado unificado.

Sin embargo, nosotros queríamos hacer distintas inteligencias; ¿cuál sería entonces la estrategia por seguir? Podríamos colocar una propiedad "tipo" en la clase y luego, en función de su valor, dividir el flujo de ejecución representando distintas lógicas:

```
if (tipo == 1)
{
    // lógica del jugador tipo A
}
else if (tipo == 2)
{
    // lógica del jugador tipo B
}
else if (tipo == 3)
{
    // lógica del jugador tipo C
}
```

¡Pero esto sería demasiado tedioso! Además, la cantidad de código en la clase crecería mucho, y si no tenemos el cuidado suficiente podremos, intentando modificar el comportamiento de un tipo de jugador, modificar el equivocado alterando lo que ya funcionaba correctamente.

Entonces, usted se podrá preguntar: ¿por qué no utilizamos el mecanismo herencia? ¡Excelente idea! Veamos: podríamos crear distintas subclases de **Jugador** que implementaran el comportamiento de cada tipo; podrían ser **Bot1** y **Bot2**.
Entonces, agregaremos a nuestro programa:

```
class Bot1 : Jugador
{
    public void RealizarMovida()
    {
        Console.WriteLine("Clase: Bot1, Método: RealizarMovida");
```

```
    }
}
class Bot2 : Jugador
{
    public void RealizarMovida()
    {
        Console.WriteLine("Clase: Bot2, Método: RealizarMovida");
    }
}
```

Ahora, nuestro diagrama quedará conformado del siguiente modo:

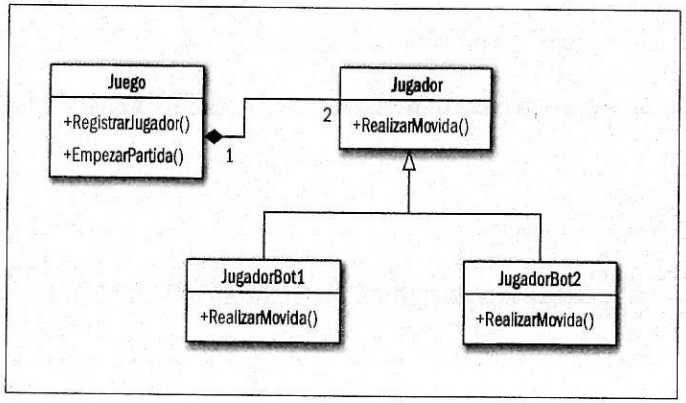


***Figura 10.** Nuestro nuevo diagrama de clases.*

Nos estamos acercando a nuestra solución, ahora deseamos que en la partida intervengan los jugadores **Bot1** y **Bot2**, para lo cual modificaremos el método **Main**:

```
class Foo
{
    static void Main(string[] args)
    {
        Bot1 jug1 = new Bot1();
        Bot2 jug2 = new Bot2();
        Juego j = new Juego();
        j.EmpezarPartida(jug1, jug2);
    }
}
```


La idea sería que ahora la instancia de la clase **Juego** invocase los métodos de los objetos del tipo **Bot1** y **Bot2**. Sin embargo, en pantalla veremos nuevamente:

```
Console.WriteLine("Clase: Jugador, Método: RealizarMovida");
Console.WriteLine("Clase: Jugador, Método: RealizarMovida");
```

¡Esto no era lo que esperábamos! Analicemos un poco qué es lo que ha ocurrido:

El método **EmpezarPartida** recibe dos objetos del tipo **Jugador**; nosotros estamos enviando un objeto del tipo **Bot1** y otro del tipo **Bot2**. El compilador no arroja error alguno, ya que **Bot1** es subclase de **Jugador**, al igual que **Bot2**, por lo que es totalmente válido, ya que al ser subclases de **Jugador** también pueden ser considerados de dicho tipo (podríamos establecer una analogía definiendo que si un gato es una particularización de un animal, entonces un gato ES un animal). Lo que aquí hicimos sin querer fue lo que se denomina *upcasting*.
Luego, el método **EmpezarPartida** invocó el método **RealizarMovida** de un objeto, que si bien es del tipo **Bot1**, fue asignado a un parámetro del tipo **Jugador**; por lo tanto, cuando se realiza la invocación, se ejecuta la versión del **Jugador**.

Para modificar este comportamiento, deberíamos declarar al método **RealizarMovida** como **virtual**; para realizar esto nos alcanzará con anteponer dicha palabra como un modificador más:

```
class Jugador
{
    public virtual void RealizarMovida()
    {
        Console.WriteLine("Clase: Jugador, Método: RealizarMovida");
    }
}
```

III ENTENDIENDO LA SOBRECARGA DE MÉTODOS

Es importante realizar pruebas de distintos ejemplos para entender bien cómo funcionan los métodos virtuales y poder anticipar con total seguridad cuál será el comportamiento de nuestro programa en cada caso. Por suerte, el excelente depurador que incluye el entorno Visual Studio .NET nos facilitará mucho la tarea, ya que podremos seguir paso a paso la ejecución de nuestro programa.

Luego, cuando sobrecarguemos dicho método en las subclases correspondientes, deberemos agregar otro modificador llamado **override**:

```
class Bot1 : Jugador
{
    public override void RealizarMovida()
    {
        Console.WriteLine("Clase: Bot1, Método: RealizarMovida");
    }
}
class Bot2 : Jugador
{
    public override void RealizarMovida()
    {
        Console.WriteLine("Clase: Bot2, Método: RealizarMovida");
    }
}
```

Si ejecutamos el programa nuevamente, ahora obtendremos la siguiente salida:

```
Console.WriteLine("Clase: Bot1, Método: RealizarMovida");
Console.WriteLine("Clase: Bot2, Método: RealizarMovida");
```

Este mecanismo, que en apariencia es sencillo, brinda al lenguaje una característica sobresaliente que, si la empleamos correctamente, otorgará a nuestros modelos elegancia y sencillez, dos de las cualidades más buscadas e importantes en un sistema.

Notemos en nuestro ejemplo que la clase **Juego** está, finalmente, invocando un método de una clase (**Bot1**, **Bot2**) que ni siquiera conoce directamente. El día de mañana podremos crear nuevas subclases de **Jugador** y hacer que **Juego** las invoque sin modificar una sola línea de esta última clase.

EL MODIFICADOR OVERRIDE

Otros lenguajes orientados a objetos, como el C++, no especifican que deba declarase un modificador override cuando se está sobrecargando un método virtual. Por suerte, C# ha incorporado esta característica, que evita posibles errores y malas interpretaciones.

Vayamos ahora un paso más adelante. Podríamos establecer que todas las implementaciones de algún jugador debieran ser subclases de **Jugador**, y de esta subclase dejar únicamente el esqueleto, ya que, al fin y al cabo, imponiendo esta regla no cabe la posibilidad de que exista una instancia **Jugador**, sino que existirán instancias de sus subclases.

"Dejar el esqueleto" implicaría colocar sólo las declaraciones de los métodos; ningún sentido tendría colocar sus definiciones, ya que nunca serán ejecutadas. Obrando de esta manera estaríamos creando lo que se conoce como interfaz (en inglés, *interface*).

Interfaces

Como mencionamos en el párrafo anterior, una interfaz es una estructura de datos que únicamente posee declaraciones, pero que no realiza implementación de ningún método. Un contrato que es expresado para que posteriormente lo implemente quien se subscriba a él.
Para convertir nuestra clase **Jugador** en una interfaz deberemos escribir:

```
interface Jugador
{
    void RealizarMovida();
}
```

Nótese que:

- Cambiamos la palabra reservada **class** por **interface**.
- Eliminamos el modificador **public** (ya que todos los métodos declarados dentro de una interfaz siempre son públicos).

MÁS ACERCA DE LAS INTERFACES

Una interfaz puede declarar propiedades (declarando el **get** y/o el **set** pero sin implementarlos). Una interfaz **NO** puede poseer variables ni constantes, ni heredar de ninguna clase, pero **SÍ** de otras interfaces. Finalmente, una clase que herede de una interfaz debe implementar **TODOS** los métodos declarados por ella.

- Eliminamos el modificador **virtual** (todos los métodos en una interfaz se comportan como virtuales).
- Eliminamos **RealizarMovida** (una interfaz sólo posee declaraciones).

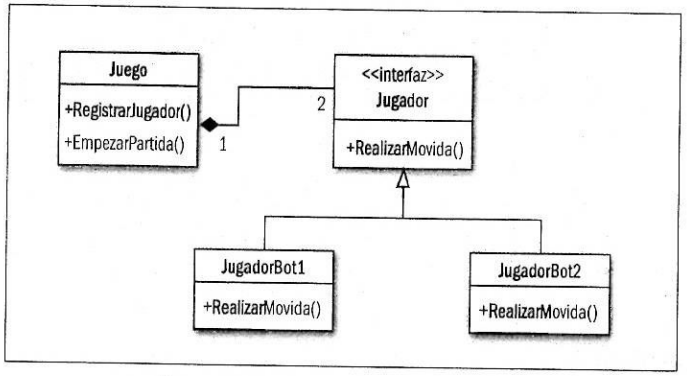


Figura 11. *Nuestro diagrama de clases final.*

Existe una convención arbitraria respecto a la creación de identificadores de las interfaces. Esta convención consiste en anteponer una **I** al nombre deseado (por lo que nuestra flamante interfaz debería llamarse **IJugador**).

Posteriormente, en las clases **Bot1** y **Bot2** hemos eliminado el **override**. Sin embargo, hubo también un cambio conceptual importante: las clases **Bot1** y **Bot2** ya no heredan de una clase, sino de una interfaz.
Es importante mencionar que el lenguaje C# **NO soporta herencia múltiple**, esto significa que una clase únicamente puede heredar de **una y sólo una** clase. Sin embargo, sí puede suscribir contratos con muchas interfaces.

El poder de una interfaz bien empleada es enorme. Cada clase que suscriba un contrato con alguna interfaz podrá luego ser tipificada con el tipo de dicha interfaz (algo así como lo que ocurrió con **Bot1**, que fue pasada como **Jugador**).

Clases abstractas

Es muy posible, en el caso anterior, especificar **Jugador** como una clase especial, denominada **clase abstracta**. Las clases abstractas poseen la característica particular de no poder ser instanciadas. Éstas son creadas con el propósito de ser utilizadas posteriormente por medio de la herencia.

¿Y cuál es la diferencia respecto a las interfaces? Las clases abstractas pueden definir métodos o declararlos como abstractos, para luego ser implementados por subcla-

ses. Además, pueden poseer variables y constantes; es decir que se asemejan mucho más a una clase convencional que las interfaces.
Veamos cómo podríamos haber declarado **Jugador** como una clase abstracta:

```
abstract class Jugador
{
    abstract public void RealizarMovida();
}
```

Como se puede apreciar, colocamos la palabra reservada **abstract** antes de **class** y antes de la declaración del método que se comportará como un método virtual convencional (por lo que habrá que colocar **override** antes de sobrecargarlo en las subclases, como habíamos hecho en un principio).

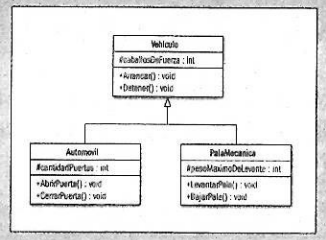
RESUMEN

En este capítulo hemos analizado conceptos en apariencia simples pero muy profundos. Utilizar correctamente el polimorfismo y la herencia será determinante en la creación de sistemas complejos. Las propiedades nos brindan la posibilidad de reforzar el encapsulamiento; la herencia nos permite crear programas con mucho menos cantidad de código y más fáciles de mantener; los métodos virtuales, interfaces y las clases abstractas nos ayudan a explotar el polimorfismo hasta el límite. Todo esto, sumado a la gran arquitectura con la cual fue creada la librería BCL. Nunca antes otro lenguaje de programación orientado a objetos había contado con un número similar de recursos. Programar en C# es realmente un placer.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Es posible definir una clase que no sea subclase de ninguna otra?



2 Una variable privada puede ser accedida desde (puede existir cero, una o varias opciones correctas):

a. La clase desde la cual fue declarada.

b. Una subclase de la clase desde la cual fue declarada.

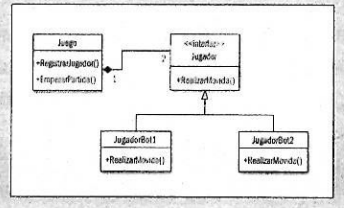
c. Externamente, desde un objeto de la clase desde la cual fue declarada.

3 ¿Cuándo es conveniente brindar acceso irrestricto a una variable miembro y cuándo conviene regular su acceso por medio de una propiedad?

4 ¿Para qué se utilizan los métodos virtuales?

5 ¿Cuáles son las diferencias entre una interfaz y una clase abstracta?

6 ¿Puede una interfaz heredar de una clase?



Capítulo 5

Colecciones

El array es uno de los recursos más poderosos que ofrece C#. Analizaremos aquí diferentes tipos de arrays, de una dimensión y multidimensionales, cómo usarlos y los beneficios que ofrecen.

Veremos qué son los indexadores, qué es una colección y cuáles son las interfaces más importantes de colecciones que posee el framework .NET (IEnumerable, ICollection, IComparable, IList).

Por último, explicaremos qué es un diccionario y el uso de la clase Hashtable.

ARRAYS

Hasta este momento hemos aprendido la manera de crear variables que contengan en su interior un valor determinado relacionado con el tipo de dato de las mismas. En el caso de que quisiéramos almacenar, por citar un ejemplo, el nombre de un alumno, podríamos especificar:

```
string nombre;
```

¿Y si quisiéramos especificar cinco de las notas que el alumno obtuvo en un determinado curso? Si nos aferráramos sólo a lo que hemos aprendido hasta el momento, deberíamos escribir:

```
float nota1;
float nota2;
float nota3;
float nota4;
float nota5;
```

Esto no es para nada práctico, debido a que todo cálculo relacionado con el conjunto de datos (como, por ejemplo, el promedio de las notas) nos obligará a escribir el nombre de cada una de las variables.

```
float promedio = (nota1 + nota2 + nota3 + nota4 + nota5) / 5;
```

Y si el día de mañana la cantidad de notas fuera diez en lugar de cinco, entonces deberíamos modificar nuestro programa del siguiente modo:

```
float promedio = (nota1 + nota2 + nota3 + nota4 + nota5 + nota6 +
nota7 + nota8 + nota9 + nota10) / 5;
```

Afortunadamente, en este tipo de casos, el lenguaje C# admite diversas maneras de lidiar con conjuntos de datos.
Un método clásico para hacerlo es por medio de arrays, que es el tipo de colección más simple que nos ofrece este lenguaje de programación.
Un **array** es un conjunto de elementos de un cierto tipo, a los cuales es posible acceder por medio de un índice. Son especialmente útiles en aquellas situaciones donde debemos guardar no un dato, sino un conjunto de datos de un mismo tipo. Un array posee un tipo y un identificador (hasta este punto, como cualquier variable). Lo que en sintaxis de declaración diferencia a un array de una variable convencional es el sufijo de corchetes (**[]**) que sigue al tipo de dato de éste.

Veamos cómo declarar un array de números flotantes:

```
float[] notas;
```

Con lo escrito, hemos declarado el array, pero aún no podremos utilizarlo. Para ello deberemos solicitar memoria para la cantidad de elementos que poseerá, y esto lo hacemos utilizando el operador **new**:

```
notas = new float [5];
```

Nótese que luego del operador **new** especificamos el tipo de dato, y esta vez entre los corchetes indicamos cuál será la cantidad de elementos de nuestro array.
También podríamos haber especificado todo en una sola línea:

```
float[] notas = new float [5];
```

A continuación, en las **Figuras 1** y **2**, se pueden obsevar los diagramas de declaración y definición de arrays:

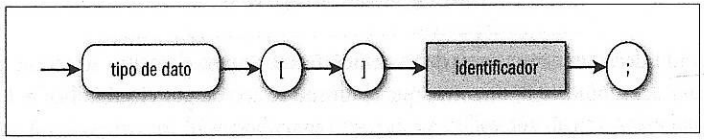


***Figura 1.** Declaración de un array.*

LOS LÍMITES DE UN ARRAY

En C#, los límites de un array son verificados (algo que no se hace en C o en C++). Si intentamos acceder a un elemento no válido (por ejemplo, mayor al número total de elementos), el framework arrojará una excepción en tiempo de ejecución.

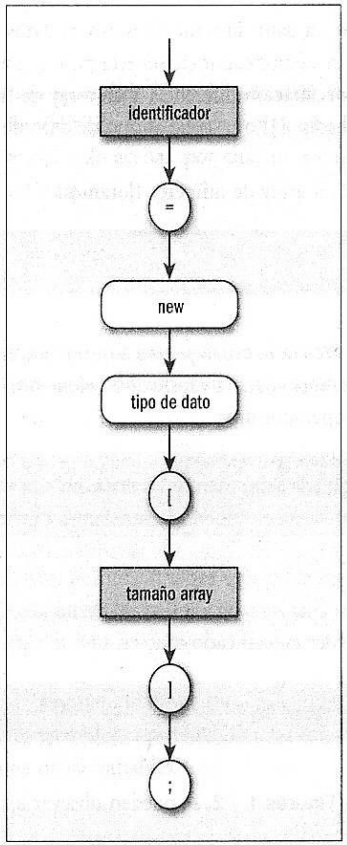


Figura 2. *Definición de un array.*

Cuando declaramos un array de **n** elementos, el primer elemento se accede por medio del subíndice **0**, mientras que el último se accede por el subíndice **n-1**.

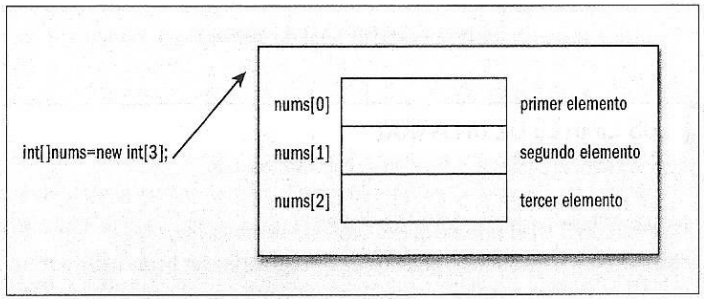


Figura 3. *Acceso a un array.*

Entonces, en un array de tres elementos, el primer elemento se accederá por el subíndice 0, el segundo por el subíndice 1 y el tercero por el subíndice 2.

Ahora podremos acceder a cada elemento del array por medio de un subíndice (**Figura 3**), por lo que podríamos escribir:

```
float promedio = (notas[0] + notas[1] + notas[2] + notas[3]
   + notas[4]) / 5;
```

Si calculamos el promedio del modo en que lo hicimos en el listado anterior, no estaremos haciendo un buen uso de los arrays.

El poder de este recurso se explota mejor utilizando sentencias de bucle.

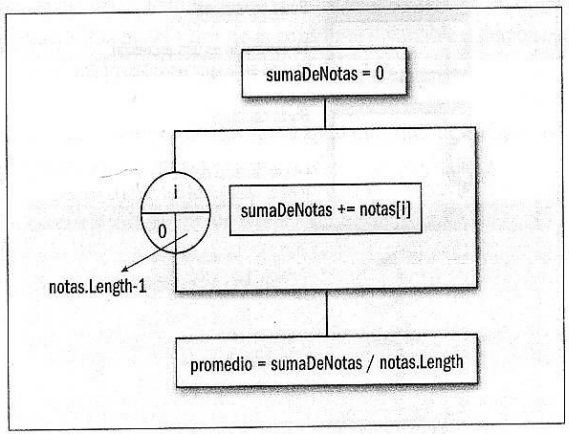


Figura 4. *Diagrama de flujo de nuestro pequeño programa.*

Con los diagramas de flujo podremos expresar, sin obligación de ligarnos al lenguaje de programación, cómo será el flujo de ejecución de nuestro programa.

III ¿PARA QUÉ SIRVE UN DIAGRAMA DE FLUJO?

Los diagramas de flujo se utilizan para diseñar algoritmos esquemáticamente, antes de llevarlos al código. De este modo, nos concentramos sólo en el problema, y no en una sintaxis particular del lenguaje de turno, porque un diagrama de flujo no está relacionado específicamente con ningún lenguaje de programación.

Existen diversos tipos de recuadros para expresar diferentes tipos de sentencias:

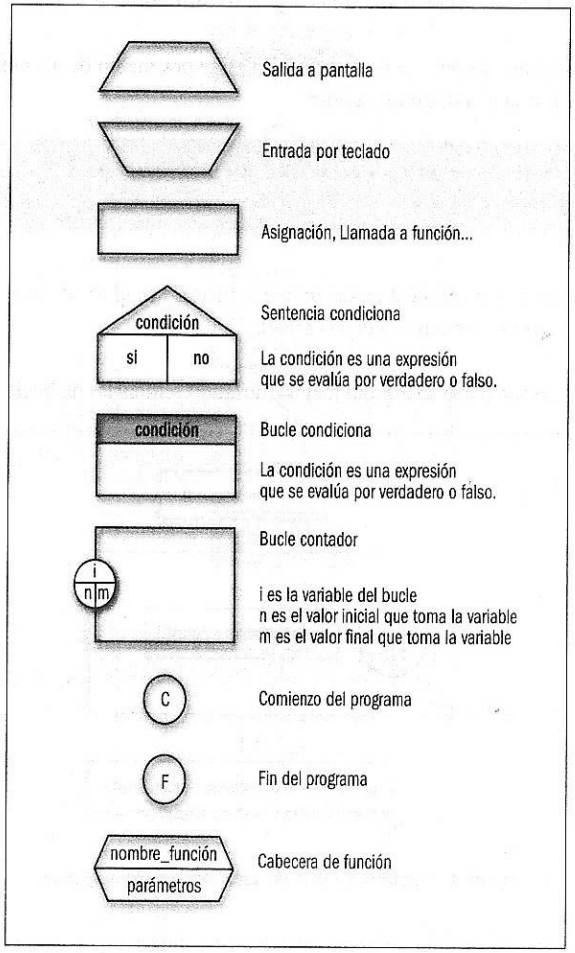


***Figura 5.** Diagramas de flujo.*

III EN C# LOS ARRAYS SON OBJETOS

¿De dónde salió la propiedad **Length**? Cada objeto declarado es descendiente de la clase **Object**. Los arrays derivan implícitamente de la clase **Array** (que desciende de la clase **Object**), lo cual brinda métodos y propiedades para su manipulación. Una de ellas es **Length**, que tiene como función devolver el total de elementos que posee el arreglo.

Y ahora, si vamos a código C#:

```
float sumaDeNotas = 0;
for (int i=0; i<notas.Length; i++)
      sumaDeNotas += notas[i];
float promedio = sumaDeNotas / notas.Length;
```

Analicemos en detalle el listado anterior:

1. Declaramos una variable donde acumularemos la sumatoria de las notas.
2. Especificamos una sentencia de bucle **for**, desde 0 hasta la cantidad máxima de elementos menos uno (cuando **i** sea igual a la cantidad máxima de elementos, no se ingresará en el cuerpo del bucle).
3. En el cuerpo del bucle acumulamos las notas en una variable. El operador **+=** (suma y asignación) suma el valor de la constante o variable **a** derecha con la variable **a** izquierda, y luego lo asigna a la variable **a** izquierda.

   Existen otros operadores que nos facilitan la escritura de código, así como éste. Veamos algunos de los más comunes:

OPERADOR	SIGNIFICADO	EJEMPLO	EXPRESIÓN EQUIVALENTE
+=	Suma y asignación	x += y;	x = x + y;
-=	Resta y asignación	x -= y;	x = x - y;
*=	Producto y asignación	x *= y;	x = x * y;
/=	Cociente y asignación	x /= y;	x = x / y;
++	Incremento	x++;	x = x + 1;
--	Decremento	x--;	x = x - 1;

***Tabla 1.** Listado de operadores más comunes.*

4. Realizamos el cálculo del promedio haciendo uso de la sumatoria calculada en la variable **sumaDeNotas** y la cantidad total de elementos del array.

Ahora, en el caso de que la cantidad de notas fuera diez en lugar de cinco, el listado anterior permanecería sin modificación alguna. De esta manera, podemos empezar a avizorar cuáles son las ventajas de este recurso.

¿Qué sucedería en el caso de que la cantidad de notas fuese mil en lugar de cinco o diez? No habría otro modo práctico de manipularlas sin hacer uso de arrays.

Métodos de la clase Array

Como hemos mencionado, todo array deriva implícitamente de la clase **Array**. Veamos cuáles son los métodos y las propiedades que podremos utilizar:

MÉTODO	ESTÁTICO	DESCRIPCIÓN
BinarySearch	Sí	Búsqueda binaria de un elemento en el array ordenado.
Clear	Sí	Limpia un array completo o en parte.
Copy	Sí	Copia la cantidad de elementos especificada de un array a otro.
CopyTo	No	Copia desde el elemento especificado de un array a otro.
GetEnumerator	No	Devuelve un IEnumerator para el array. Dicho objeto, como veremos más adelante, lo utilizaremos para navegar el array.
GetLength	No	Devuelve el tamaño del array.
GetValue	No	Devuelve un elemento en el índice especificado por parámetro.
Reverse	Sí	Invierte el orden de los elementos de un array.
SetValue	No	Fija un elemento del array en el índice especificado por parámetro.
Sort	Sí	Ordena el contenido de un array.

***Tabla 2.** Métodos utilizados en la clase Array.*

PROPIEDAD	DESCRIPCIÓN
IsFixedSize	Indica si el array posee un tamaño fijo.
IsReadOnly	Indica si el array es de sólo lectura.
IsSynchronized	Indica si el array puede ser accedido de modo seguro por diversos threads en modo concurrente.
Length	Tamaño del array.
Rank	Indica la cantidad de dimensiones de un array.

***Tabla 3.** Propiedades de la clase Array.*

Veamos cómo utilizar algunos de estos métodos y propiedades:

Crearemos un array de diez elementos con valores de elementos arbitrarios; luego ordenaremos su contenido y buscaremos un valor en particular. El resultado de la búsqueda será mostrado en pantalla.

UN ARRAY EN MEMORIA

El espacio en memoria que ocupa un array corresponde al espacio que ocupa un elemento del tipo de dato al cual pertenece, multiplicado por el tamaño del array. Si cada elemento ocupa 4 bytes, un array de 10 elementos ocupará 40 bytes. Los elementos de un array son colocados adyacentemente en la memoria, para acceder a ellos fácil y rápidamente por medio de un subíndice.


```
int[] nums = new int[10];
nums[0] = 3;
nums[1] = 1;
nums[2] = 8;
nums[3] = 5;
nums[4] = 9;
nums[5] = 2;
nums[6] = 0;
nums[7] = 6;
nums[8] = 12;
nums[9] = 10;
// Ordeno el array
Array.Sort(nums);
// Realizo una búsqueda sobre el mismo, de un
// elemento arbitrario
int elemento = Array.BinarySearch(nums, 2);
if (elemento >= 0)
      Console.WriteLine("Elemento encontrado en índice: {0}",
         elemento);
else
     Console.WriteLine("Elemento no encontrado");
```

Del listado anterior es interesante notar:

- La invocación al método **Sort** la realizamos por medio de la clase **Array** y no por medio del objeto array **nums**, debido a que es un método estático. Lo mismo sucede con **BinarySearch**.
- El método **BinarySearch** recibe como primer parámetro el array por ordenar, y como segundo parámetro, el elemento por buscar (aunque este método se encuentra sobrecargado y existen otras variantes). Devuelve el índice del elemento encontrado, y en caso contrario, un valor negativo.

Ahora veremos otro ejemplo. Crearemos un array de números arbitrarios. Copiaremos el contenido de dicho array en otro e invertiremos su contenido. Finalmente mostraremos el contenido de ambos arrays en pantalla.

```
int[] nums = new int[5];
nums[0] = 3;
nums[1] = 1;
```

```
nums[2] = 8;
nums[3] = 5;
nums[4] = 9;
int[] nums2 = new int[5];
// Copio el contenido de un array a otro
nums.CopyTo(nums2, 0);
// Invierto uno de los arrays
Array.Reverse(nums2);
Console.WriteLine("Array notas");
for (int i=0; i<nums.Length; i++)
     Console.WriteLine("elemento {0}: {1}", i, nums[i]);
Console.WriteLine("Array notas2");
for (int i=0; i<nums2.Length; i++)
     Console.WriteLine("elemento {0}: {1}", i, nums2[i]);
```

Del listado anterior es interesante notar:

- La invocación del método **CopyTo** la realizamos haciendo uso del array **nums**, debido a que éste no es estático (el método **Copy** sí lo es). El primer parámetro que espera **CopyTo** es el array destino, y el segundo parámetro, el índice a partir del cual comenzar la copia.
- El método **Reverse** es estático, por lo tanto lo invocamos por medio de la clase **Array**.

Cabe destacar, como ya mencionamos, que muchos de los métodos de la clase **Array** (así como muchos métodos de las clases de la librería **BCL**) se encuentran sobrecargados, por lo que no existe un solo modo de utilizarlos. En este caso hicimos uso de la "versión" de los métodos que mejor se adaptaba a nuestras necesidades.

Arrays como parámetros de métodos

Los arrays son **tipos de datos referenciados**, al igual que las clases. Cuando pasemos un array como parámetro, no estaremos pasando una copia de éste, sino una referencia a él, lo que permite a una función modificar directamente su contenido. Veamos un ejemplo de un método que recibe un array como parámetro:

```
void ModificarArray(int[] nums)
{
    nums[0] = 50;
}
```


Nótese que el parámetro es del tipo **int[]** y que no especifica el tamaño del arreglo, es decir que el método podrá recibir arrays de enteros de cualquier tamaño. En todo caso sería conveniente verificar, antes de a leer o escribir un elemento, si el índice se encuentra dentro del rango válido (por medio del uso de la propiedad **Length**).

Veamos el programa completo:

```
class Foo
{
    public static void ModificarArray(int[] arr)
    {
        arr[0] = 50;
    }
    static void Main(string[] args)
    {
        int[] nums = new int[5];

        nums[0] = 3;
        nums[1] = 1;
        nums[2] = 8;
        nums[3] = 5;
        nums[4] = 9;
        Console.WriteLine("Array notas");
        for (int i=0; i<nums.Length; i++)
            Console.WriteLine("elemento {0}: {1}", i, nums[i]);
        Foo.ModificarArray(nums);

        Console.WriteLine("Array notas modificado");
        for (int i=0; i<nums.Length; i++)
            Console.WriteLine("elemento {0}: {1}",
                i, nums[i]);int[] nums2 = new int[5];
    }
}
```

La línea de mayor interés para nosotros es aquélla en la que invocamos el método estático **ModificarArray** (es estático pero podría no serlo). Lo importante es ver que para pasar el array como parámetro simplemente escribimos su nombre sin hacer uso de los corchetes y sin ningún modificador. Si, en el caso contrario, deseamos pasar en forma de parámetro **un elemento** del array, deberemos escribir:

```
Foo.ModificarElemento(ref nums[0]);
```

El método **ModificarElemento** no recibe un array, sino el elemento de un array, por lo que el tipo de dato debe ser igual al tipo de dato del array (en nuestro caso, simplemente un entero):

```
void ModificarElemento(ref int num)
{
    num = 33;
}
```

En realidad, el método **ModificarElemento** recibe un número entero que puede ser elemento de un array o una variable convencional. Es importante entender que los elementos de un array pueden ser vistos como variables convencionales del tipo que corresponde al arreglo.

El hecho de utilizar **ref** como modificador del parámetro es para especificar que dicho parámetro debe ser pasado como referencia. Si bien los arrays son tipos de datos referenciados, cuando se pasan elementos de ellos se envían copias y no referencias.

También podemos crear otro método que reciba un número entero sin que éste sea referencia, y luego enviarle el elemento de un array:

```
public static void MostrarValor(int num)
    {
        Console.WriteLine("valor variable: {0}", num);
    }
```

Y ahora el listado completo, en la página siguiente:

III ARRAY QUE CONTIENE DIFERENTES TIPOS DE DATOS

Para crear un array que pueda contener objetos de diverso tipo, deberemos crear un array de objetos de tipo **Object**. Como todo tipo de dato es descendiente de **Object**, podremos agregarle números, strings, booleanos o cualquier objeto creado por nosotros.


```
class Foo
{
    public static void ModificarElemento(ref int num)
    {
        num = 33;
    }
    public static void MostrarValor(int num)

    {
        Console.WriteLine("valor variable: {0}", num);
    }
    static void Main(string[] args)
    {
        int[] nums = new int[5];

        nums[0] = 3;
        nums[1] = 1;
        nums[2] = 8;
        nums[3] = 5;
        nums[4] = 9;
        Foo.ModificarElemento(ref nums[0]);
        Foo.MostrarValor(nums[0]);
        Console.WriteLine("Array notas");
        for (int i=0; i<nums.Length; i++)
            Console.WriteLine("elemento {0}: {1}", i, nums[i])
        int variableConvencional = 10;
        Foo.ModificarElemento(ref variableConvencional);
        Foo.MostrarValor(variableConvencional);
    }
}
```

Dedíquese especial atención, en el listado anterior, a que **enviar un elemento** de un array de enteros supone, en cuanto a resultado, exactamente **lo mismo que enviar una variable** del tipo que corresponda al array.

Valores predeterminados en un array

Los arrays de tipos de datos que el compilador maneja por valor son inicializados automáticamente cuando son definidos:

```
int[] nums = new int[5];
```

En el ejemplo anterior, el array comenzará relleno con ceros.
Pero si el array es de un tipo referenciado (array de objetos), entonces los elementos son inicializados en **null**, y para poder hacer uso de cada uno de ellos deberemos inicializarlos individualmente por medio del operador **new**. Veamos un ejemplo de esto último para que quede más claro.

Si tenemos la clase **Foo** definida del siguiente modo:

```
class Foo
{
    Foo()
    {
        Console.WriteLine("Constructor");
    }
}
```

Podríamos crear un array de objetos de ésta escribiendo:

```
Foo[] arrayDeFoo = new Foo[5];
```

Sin embargo, esto no producirá la invocación de los constructores de los objetos de **Foo**. El array tendrá cinco elementos con valor **null**, que son referencias a objetos del tipo **Foo** que aún no han sido inicializadas.
Para poder utilizar los elementos deberemos inicializarlos en forma individual. Podríamos realizar esta tarea inmediatamente después de la declaración del array o a medida que lo requiramos (hay que tener en cuenta que inicializar un objeto implica ejecutar código de constructores; realizar esto una gran cantidad de veces podría influir en la performance de nuestra aplicación).

```
for (int i=0; i<arrayDeFoo.Length; i++)
    arrayDeFoo[i] = new Foo();
```

Ahora sí, se ha ejecutado el constructor de cada uno de los elementos del array, tras lo cual podremos acceder a sus variables, métodos y propiedades. En la **Figura 6** se puede apreciar otro ejemplo de lo mencionado:

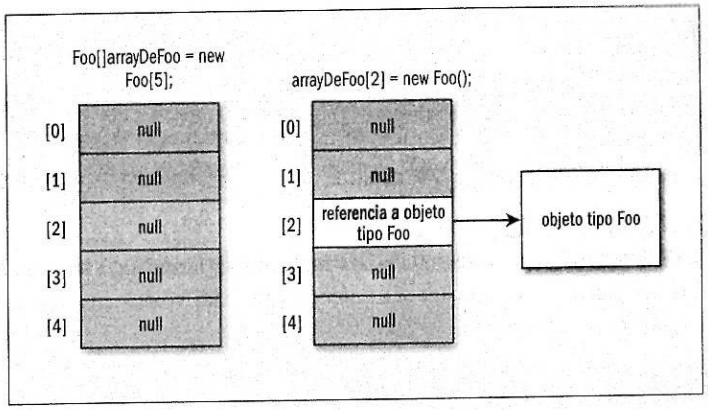


***Figura 6.** Inicialización de arrays.*

Inicializando elementos de arrays

Es posible dar valores arbitrarios a los elementos de un array de un modo más abreviado que el que hemos empleado hasta el momento.
Para ello, en lugar de hacer:

```
int[] nums = new int[5];
nums[0] = 3;
nums[1] = 1;
nums[2] = 8;
nums[3] = 5;
nums[4] = 9;
```

podremos hacer simplemente:

```
int[] nums = new int[5] {3, 1, 8, 5, 9};
```

EL TAMAÑO DE LOS ARRAYS

Los arrays que estamos analizando en estas páginas no pueden ser redimensionados, es decir que si le asignamos un tamaño específico, dicho tamaño será fijo hasta que el array sea destruido. C# cuenta con una implementación de arrays redimensionables en la clase **ArrayList** de la librería BCL.

O mejor aún:

```
int[] nums = {3, 1, 8, 5, 9};
```

Recorriendo arrays

Hasta el momento, hemos recorrido arrays por medio de la sentencia **for**. También podríamos haberlo hecho utilizando la sentencia **while** para recorrer un array, tal como se detalla a continuación:

```
int[] nums = new int[5];
int i=0;
while (i<nums.Length)
{
    Console.WriteLine(nums[i]);
    i++;
}
```

Sin embargo, la sentencia que mejor se adecua a los arrays tal vez sea el **foreach** (ausente en los lenguajes C y C++). En el **Capítulo 2** ya habíamos introducido esta sentencia; aquí veremos nuevamente cómo recorrer un array haciendo uso de ella:

```
int[] nums = {10, 20, 30, 40, 50}
foreach(int i in nums)
{
    Console.WriteLine(i);
}
```

Como podemos apreciar, en la sentencia **foreach** declaramos una variable que oficiará de elemento en cada iteración. Luego utilizamos dicha variable para mostrar en pantalla el contenido del array (tengamos en cuenta que con dicha variable podremos leer el contenido del array, pero no podremos modificar el contenido de los elementos).

El modificador params

Es posible especificar un modificador al parámetro de una función llamado **params**, que nos permitirá pasar arrays o un conjunto de elementos que el compilador traducirá a arrays de manera dinámica. Para ello deberemos escribir:


```
void Listar(params int[] nums)
{
    for (int i=0; i<nums.Length; i++)
        Console.WriteLine(nums[i]);
}
```

Veamos ahora cómo invocar el método **Listar**, por medio de un array convencional:

```
int[] nums = {3, 1, 8, 5, 9};
foo.Listar(nums);
```

O un array creado dinámicamente a partir de un conjunto de elementos:

```
foo.Listar(1, 2, 5, 10, 99);
```

ARRAYS MULTIDIMENSIONALES

C# soporta la definición de arrays de múltiples dimensiones. Hasta el momento, todos nuestros arrays eran de una dimensión, pero es posible definir arrays de dos dimensiones (ideales para representar tablas, cuadros de dos entradas, matrices, etc.) o de las dimensiones que nos parezcan.

Veamos cómo declarar un array de dos dimensiones:

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

Como podemos apreciar en el listado anterior, luego del tipo de dato del array entre los corchetes, existe una coma. También, en lugar de haber un solo número en la indicación del tamaño del array, figuran dos, correspondientes a la cantidad de elementos por cada dimensión.

Es muy práctico imaginar un array de dos dimensiones como una tabla, siendo la primera dimensión la cantidad de filas, y la segunda dimensión, la cantidad de columnas (o al revés, según la convención que se desee emplear).

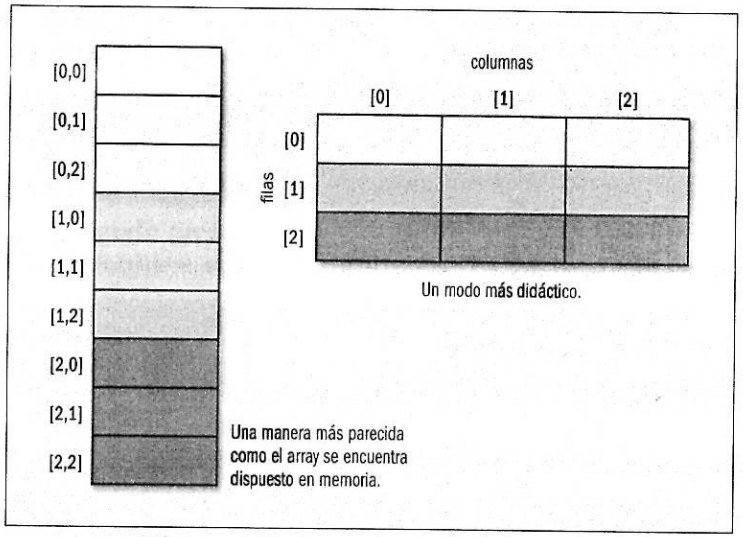


Figura 7. *Dos maneras distintas de pensar un array de dos dimensiones.*

Para poder acceder a los elementos de un array de dos dimensiones, se deberá hacer uso de la siguiente sintaxis:

```
tabla[0, 0] = 10;
```

En esta oportunidad, hemos decidido asignar el número 10 al primer elemento del array (fila 0, columna 0).
En el siguiente diagrama se puede observar cómo seguimos trabajando con otros elementos del array:

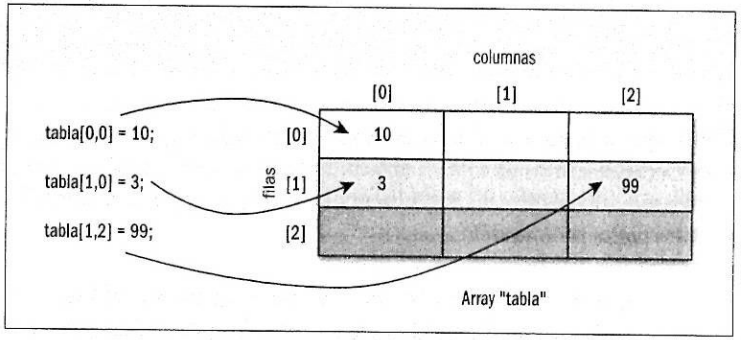


Figura 8. *Accediendo a otros elementos del array.*

Es bueno tener presente que la cantidad de dimensiones de un array sólo facilita la sintaxis para el acceso a elementos. Toda implementación en arrays multidimensionales puede realizarse en arrays de una dimensión.
El siguiente diagrama nos sirve para graficar el ejemplo:

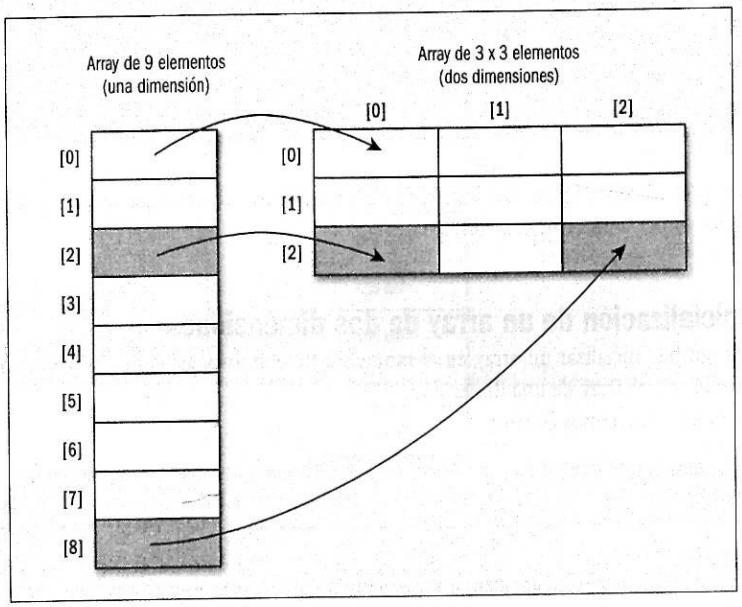


Figura 9. *Equivalencias entre un array de una dimensión y un array de dos dimensiones.*

Veamos ahora esta equivalencia en código. Si tuviéramos que representar una tabla con un array de dos dimensiones, procederíamos del siguiente modo:

```
int cantidadFilas = 3;
int cantidadColumnas = 3;
int[,] tabla = new int[cantidadFilas, cantidadColumnas];
for (int i=0; i<cantidadFilas * cantidadColumnas; i++)
     tabla[i / cantidadColumnas, i % cantidadColumnas] = i;
for (int fila=0; fila<cantidadFilas; fila++)
     for (int col=0; col<cantidadColumnas; col++)
          Console.WriteLine(tabla[fila, col]);
```

Si, en oposición al caso anterior, nos encontráramos en la obligación de representar una tabla con un array de una dimensión, entonces:

```
int cantidadFilas = 3;
int cantidadColumnas = 3;
int[] tabla = new int[cantidadFilas * cantidadColumnas];
for (int i=0; i<cantidadFilas * cantidadColumnas; i++)
    tabla[i] = i;
for (int fila=0; fila<cantidadFilas; fila++)
    for (int col=0; col<cantidadColumnas; col++)
    Console.WriteLine(tabla[fila*cantidadColumnas+col]);
```

Como podemos apreciar en el listado anterior, llevamos a cabo la conversión de una a dos dimensiones con una simple cuenta.

Inicialización de un array de dos dimensiones

Es posible inicializar un array en el momento de definirlo, así como lo habíamos hecho con el array de una dimensión.
Para esto deberemos escribir:

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

En este caso, el compilador entiende, en función de cómo dispusimos las constantes (tres grupos de tres números), que la matriz será de 3 x 3.

ARRAYS DE ARRAYS

Además de los arrays multidimensionales que analizamos en los párrafos anteriores y que son conocidos con el nombre de **arrays rectangulares**, también tenemos la posibilidad de declarar **arrays de arrays** (más frecuentemente conocidos con la denominación de *jagged arrays*).
Para declarar un array de array deberemos escribir:

```
int[][] arrayDeArrays;
```

Notemos que, ahora, dentro del corchete no hay nada, pero existe otro juego de ellos tras el primero. Esto quiere decir que poseeremos un array en el cual cada uno de sus elementos será, a la vez, otro array.

Ahora definamos el array de nivel más alto, el que contendrá al resto de los arrays:

```
arrayDeArrays = new int[5][];
```

Aquí obtendremos un array de cinco arrays. Los arrays que se encuentran dentro del primero aún no fueron definidos, por lo que lo que tendríamos en memoria se corresponde con el siguiente diagrama:

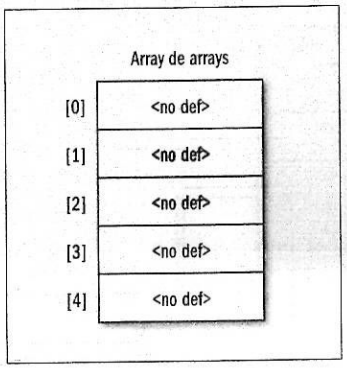


Figura 10. *Todavía no hemos definido los arrays de segundo nivel.*

Definamos, entonces, los arrays que se encuentran dentro del primer array. Éstos no tendrán por qué poseer el mismo tamaño:

```
arrayDeArrays[0] = new int[3];
arrayDeArrays[1] = new int[2];
arrayDeArrays[2] = new int[5];
arrayDeArrays[3] = new int[4];
arrayDeArrays[4] = new int[2];
```

III USO DE LOS JAGGED ARRAYS

Debido a que los jagged arrays no tienen por qué ser rectangulares (es decir que cada elemento del primer array no tiene por qué poseer un array del mismo tamaño que el elemento anterior), suelen ser más prácticos y hacer un mejor uso de recursos para casos específicos.

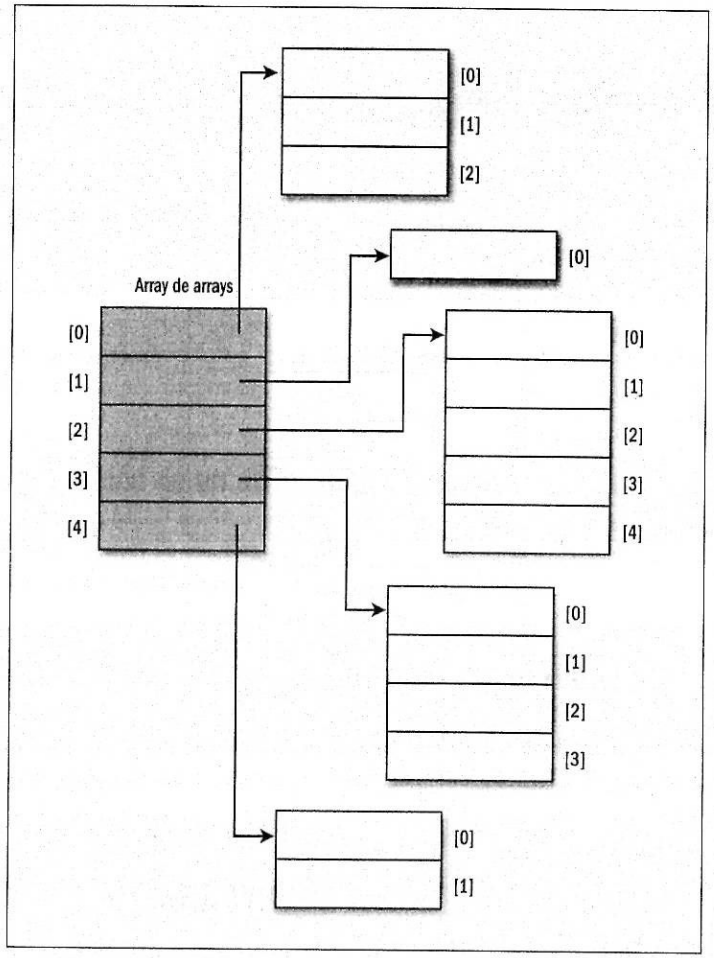


***Figura 11.** El array de arrays completo.*

También cambia ligeramente el modo en que accedemos a sus elementos:

```
arrayDeArrays[0][0] = 1;
arrayDeArrays[0][1] = 2;
arrayDeArrays[0][2] = 3;
arrayDeArrays[1][0] = 1;
arrayDeArrays[1][1] = 2;
arrayDeArrays[2][0] = 1;
arrayDeArrays[2][1] = 2;
arrayDeArrays[2][2] = 3;
arrayDeArrays[2][1] = 4;
arrayDeArrays[2][2] = 5;
arrayDeArrays[3][0] = 1;
arrayDeArrays[3][1] = 2;
arrayDeArrays[3][2] = 3;
arrayDeArrays[3][1] = 4;
arrayDeArrays[4][0] = 1;
arrayDeArrays[4][1] = 2;
```

ELEMENTOS NULOS

Es posible que algunos elementos de nuestro jagged array posean un valor nulo, es decir que no contengan ningún otro array. Ante esta posibilidad, sólo deberemos tener en cuenta que nuestro programa verifique la validez del elemento –que no sea nulo– antes de acceder a él.

INDEXADORES

En ocasiones puede ser beneficioso permitir acceder a una colección dentro de un objeto que hayamos creado como si éste fuese un array, es decir, utilizando el operador de indexación **[]**.

C# permite realizar esto no por medio de la sobrecarga del operador **[]**, como ocurre en C++, sino a través de una característica denominada **indexador**.

Un indexador se asemeja en gran medida a una propiedad, pero no posee nombre, sino que, por el contrario, es invocado cuando se utiliza el operador de indexación (**[]**) junto con el identificador del objeto en cuestión.

USO DE INDEXADORES

Los indexadores pueden ser el recurso ideal para dotar a nuestros objetos de un fácil e intuitivo método de acceso a elementos contenidos en ellos.

Supongamos que hemos implementado una clase llamada **Cadena**, que facilita el uso y la manipulación de strings (en C#, esto sería realmente innecesario, porque cuenta con un excelente soporte de strings). Nuestra clase **Cadena** poseerá como variable privada un array de caracteres donde se almacenará el texto correspondiente:

```
class Cadena
{
    private char[] texto;
}
```

Ahora bien, si deseáramos leer y fijar el contenido de la cadena deberíamos escribir:

```
public string Texto
{
    get
    {
        string str = "";
        foreach(int c in texto)
            str += (char) c;
        return str;
    }
    set
    {
        texto = new char[value.Length];
        for (int i=0; i<value.Length; i++)
            texto[i] = value[i];
    }
}
```

Analicemos lo escrito en el método de acceso **get** de la propiedad **Texto**:

1. Creamos una variable local de tipo string.
2. Accedimos a cada elemento del array de caracteres y lo copiamos sobre el string.
3. Devolvimos el string como valor de retorno del método.

En el método de acceso **set**:

1. Creamos un array del tamaño adecuado para contener el string fijado (en la palabra reservada **value**). Nótese que la segunda vez que se fije un valor al texto, se creará otro array; el primero será destruido por el recolector de basura cuando al framework le parezca adecuado (podríamos haber verificado si realmente era necesario crear otro array; o crear uno un poco más grande de lo necesario para no solicitar memoria una y otra vez, pero deseamos mantener el ejemplo sencillo).
2. Trasladamos cada carácter del string a un elemento de nuestro array.

Siguiendo con nuestro ejemplo, ahora deseamos permitir que el usuario de nuestro objeto de tipo **Cadena** pueda acceder a cada letra del string (elemento de nuestro array) como si nuestro objeto entero fuese un array. Para esto crearemos un indexador:

```
public char this[int index]
{
    get
    {
        if (index < texto.Length)
            return texto[index];
        else
            return '\0';
    }
    set
    {
        if (index < texto.Length)
            texto[index] = value;
    }
}
```

A primera vista, nuestro indexador es muy similar a una propiedad, pero si observamos bien, notaremos que el identificador es diferente. La palabra reservada **this** hace referencia al propio objeto en el cual estamos. Luego, entre corchetes, declaramos una variable que será la que podremos usar dentro de los métodos de acceso **get** y **set**, y la que contendrá el número que entre corchetes especifique el índice.

III USO DE INDEXADORES EN LA BCL

Muchas de las clases que manejan conjuntos de datos en la librería BCL utilizan indexadores. Por ejemplo, la clase String implementa indexadores para el acceso a los caracteres de la cadena de texto (sólo en modo lectura).

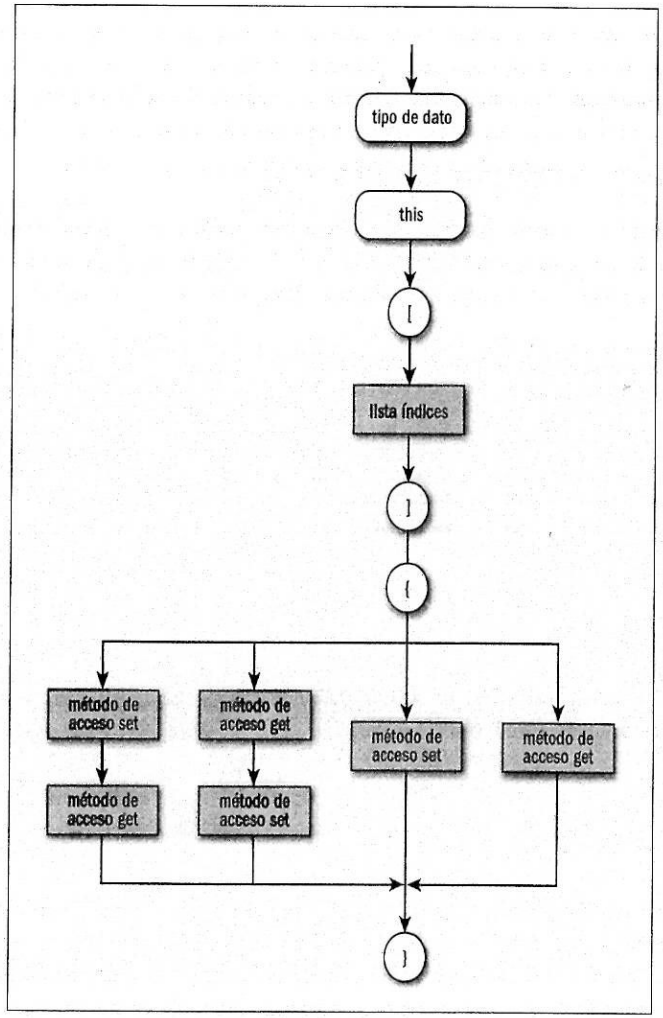


***Figura 12.** Diagrama de declaración de un indexador.*

USO ESPECIAL DE INDEXADORES

Un indexador no tiene por qué esperar sólo números enteros; es un recurso muy versátil que podría recibir cualquier tipo de dato.

Teniendo todo esto en cuenta, comprender el código escrito en los métodos de acceso **get** y **set** resultará bastante sencillo.

En el método get:

1. Verificamos si el índice especificado se encuentra ubicado dentro del rango válido de nuestro array.
2. Si es así, accedemos al elemento especificado usando la variable privada **texto**.
3. Si no es así, devolvemos un carácter nulo.

En el método set:

1. Verificamos si el índice especificado se encuentra ubicado dentro del rango válido de nuestro array.
2. En caso de comprobar que sea así, modificamos su contenido con el carácter recibido en la palabra reservada **value**.

Finalmente, sobrecargamos el método **ToString** para especificar qué string debe devolverse como representación del objeto cuando se lo requiera.

Ahora, veamos un programa sencillo que hace uso de nuestra clase **Cadena**:

```
Cadena cad = new Cadena();
cad.Texto = "el lenguaje C#";
// El objeto como string
Console.WriteLine(cad);
// Dejo una línea libre
Console.WriteLine();
// El objeto como array de caracteres
for (int i=0; i<cad.Largo; i++)
     Console.Write(cad[i]);
// Dejo una línea libre
Console.WriteLine();
// Modifico parte del contenido del texto
cad[0] = 'E';
cad[3] = 'L';
// El objeto como array de caracteres
for (int i=0; i<cad.Largo; i++)
     Console.Write(cad[i]);
```

En el listado anterior, hacemos uso del indexador definido en la clase con la finalidad de leer los caracteres del array que se encuentra encapsulado en su interior, así como para modificar su contenido.

Indexadores para acceder a arrays de dos dimensiones

Es posible especificar indexadores para acceder a arrays de más de una dimensión. Veamos un ejemplo supersencillo para entender bien esto. Queremos permitir que el siguiente código sea válido:

```
Foo f = new Foo();
f[3, 4] = 10;
Console.WriteLine(f[3, 4]);
```

Para esto, debemos especificar un indexador que acepte dos índices:

```
public int this[int idx1, int idx2]
{
    get
    {
        Console.WriteLine("método get. Indice 1: {0}, Indice 2:
        {1}", idx1, idx2);
        return 0;
    }
    set
    {
        Console.WriteLine("método set. Indice 1: {0}, Indice 2: {1}
        Valor: {2}", idx1, idx2, value);
    }
}
```

INDEXADORES CON ELEMENTOS MÚLTIPLES

El hecho de que el indexador permita recibir más de un tipo de dato no significa que debamos utilizarlo solamente para el acceso a arrays multidimensionales. Tampoco es necesario que todos los elementos que reciba el indexador sean del mismo tipo. Como hemos mencionado antes, es un recurso muy flexible.

Como se puede apreciar, ahora entre los corchetes de la primera línea de la declaración del indexador existen dos variables declaradas, y no sólo una. Estas variables representan los dos índices que se le pasarán al objeto.
También conviene destacar que nuestro indexador recibe y entrega números enteros, y no caracteres. En realidad, podría recibir y entregar cualquier tipo de dato.

COLECCIONES

Una colección es un objeto que contiene un grupo de objetos afines. Mediante una colección es posible actuar sobre la totalidad de un subconjunto de objetos.
C# posee diversas interfaces de colecciones que nuestras clases podrán implementar y, de este modo, otorgar mayor facilidad de uso.

IEnumerable y IEnumerator

Por ejemplo, vimos que es posible recorrer un array mediante la sentencia de bucle **foreach**. ¿Cómo hacer para que un objeto definido por nosotros también pueda ser recorrido de igual modo? Para esto deberemos implementar la interfaz **IEnumerable**.
Como toda interfaz, **IEnumerable** declara una serie de métodos que quien suscriba un contrato con ella deberá implementar (todos y cada uno de ellos).
En este caso particular, es sólo un método con el siguiente prototipo:

```
IEnumerator GetEnumerator();
```

IEnumarator es otra interfaz que bien podrá implementar la misma clase que implementa **IEnumerable**, o alguna otra clase afín.

IEnumerator declara la siguiente propiedad:

```
object Current {get;}
```

Esta propiedad será utilizada cuando se desee acceder al elemento actual apuntado por el enumerador. Y también declara los siguientes métodos:

```
bool MoveNext();
```

MoveNext traslada el cursor al siguiente elemento.
Reset devuelve el cursor al inicio de la colección.

```
void Reset();
```

La **Figura 13** muestra el diagrama de clases que implementaremos a continuación:

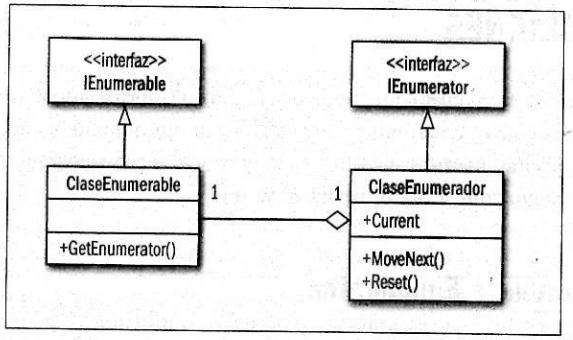


***Figura 13.** Nuestro diagrama de clases.*

Una vez que hemos implementado nuestro pequeño programa, será posible instanciar la clase **ClaseEnumerable** y recorrerla por medio de una sentencia **foreach**, tal como se muestra a continuación:

```
ClaseEnumerable objEnum = new ClaseEnumerable();
foreach (int i in objEnum)
    Console.WriteLine(i);
```

Observemos, en el listado que se encuentra a continuación, la manera en que deberemos implementar estas clases y qué código deberemos ingresar en cada uno de los métodos sobrecargados:

¿QUÉ ES UN ENUMERADOR?

Conceptualmente, un enumerador es un objeto que apunta a un elemento de una colección. Es un cursor que nos permite recorrer una lista pasando de un elemento a otro mediante un método. Los enumeradores suelen ser muy útiles, ya que permiten recorrer una colección de un mismo modo, más allá de cómo estén implementados (sobre arrays, listas enlazadas, archivos, etc.).

La clase **ClaseEnumerable:**

```
class ClaseEnumerable : IEnumerable
{
    // Lista de enteros que recorreremos
    private int[] lista = {0, 1, 2, 3, 4, 5};
    // Método GetEnumerator
    public IEnumerator GetEnumerator()
    {
        return new ClaseEnumerador(this);
    }

    // Indexador para facilitar el acceso
    // a elementos del array encapsulado
    public int this[int idx]
    {
        get
        {
            return lista[idx];
        }
    }

    // Propiedad Length que a su vez invoque la propiedad
    // Length del array encapsulado
    public int Length
    {
        get
        {
            return lista.Length;
        }
    }
}
```

La clase **ClaseEnumerador:**

```
class ClaseEnumerador : IEnumerator
{

    // Referencia al objeto por recorrer
```

```
    private ClaseEnumerable objEnum = null;
    // Elemento actual

    private int elementoActual = -1;
    // Constructor de la clase, recibe como parámetro
    // una referencia al objeto que debe recorrer
    public ClaseEnumerador(ClaseEnumerable obj)
    {
        objEnum = obj;
    }
    // Propiedad Current que retorna elemento actual
    public object Current
    {
        get
        {
            return objEnum[elementoActual];
        }
    }
    // Método MoveNext
    public bool MoveNext()
    {
        if (elementoActual < objEnum.Length-1)
        {
            elementoActual++;
            return true;
        }
        else
            return false;
    }
    // Método Reset
    public void Reset()
    {
        elementoActual = -1;
    }
}
```

ICollection

Existe otra interfaz de colecciones que se denomina **ICollection**. Esta interfaz declara las propiedades que enumeramos a continuación:

- **int Count {get;}**: devuelve la cantidad total de elementos que posee la colección.
- **bool IsSynchronized {get;}**: indica si el acceso al objeto es seguro por medio de diferentes threads concurrentes.
- **object SyncRoot {get;}**: esta propiedad devuelve un objeto que permita sincronizar el acceso a la colección.

La interfaz declara, además, el siguiente método:

```
void CopyTo(Array array, int index);
```

El método **CopyTo** copiará los elementos de la colección instancia al array recibido como parámetro desde el elemento indicado por **index**.

Si nuestra clase implementa **ICollection**, entonces los objetos de ésta podrán copiar elementos de una a otra haciendo uso del método **CopyTo**.

IComparable

Si nuestra clase implementa la interfaz **IComparable**, entonces podrá ser ordenada por medio del método **Sort**, ya que es posible determinar si un elemento es menor, igual o mayor a otro.
La interfaz sólo declara el siguiente método:

```
int CompareTo(object obj);
```

El método **CompareTo** recibe un objeto como parámetro. Se espera que el método devuelva -1 si la instancia es menor al objeto, 0 si los dos objetos son iguales y 1 si el objeto es mayor a la instancia.

Téngase en cuenta que el objeto debería ser, en realidad, del tipo de la clase que estamos comparando. Para acceder a los elementos de ésta deberemos llevar a cabo un *upcasting*, esto significa convertir el objeto recibido del tipo **object** al tipo de la clase en la cual estamos implementando.

IList

Si nuestra clase implementa la interfaz **IList**, entonces sus elementos podrán ser accedidos individualmente por medio de un índice.

La interfaz declara las siguientes propiedades:

- **bool IsFixedSize {get;}**: esta propiedad nos indica si la lista es de un tamaño fijo o si puede crecer dinámicamente.
- **bool IsReadOnly {get;}**: indica si la lista es de sólo lectura.
- **object this[int index]] {get; set;}**: este indexador nos brinda la posibilidad de leer y modificar los elementos de la lista.

Por otro lado, la interfaz declara los siguientes métodos:

- **int Add(object value);**: agrega elementos a la lista.
- **void Clear();**: limpia la lista.
- **bool Contains(object value);**: indica si un elemento se encuentra contenido en la lista o no.
- **int IndexOf(object value);**: devuelve el índice al elemento pasado como parámetro, y -1 si éste no se encuentra dentro de ella.
- **void Insert(int index, object value);**: este método se encarga de insertar un elemento en una posición determinada.
- **void Remove(object value);**: elimina un elemento de la lista.
- **void RemoveAt(int index);**: elimina un elemento de la lista especificando su posición dentro de ésta.

DICCIONARIOS

Un diccionario es una colección que asocia un valor a una clave. Podría ser considerado como un array que utiliza índices no numéricos para acceder a los elementos. Esto es muy útil en innumerables ocasiones; un ejemplo podría ser querer obtener la referencia a un objeto a partir de su nombre:

```
Sprite nave = SpriteManager["nave enemiga jefe"];
```

¡MÁS INFORMACIÓN!

Este libro no pretende reemplazar la completísima ayuda que ofrece el sitio **MSDN (http://msdn.microsoft.com)** o la que viene incluida con el entorno de desarrollo integrado **Microsoft Visual Studio .NET**. Dicho recurso es ideal para obtener información de referencia respecto a clases, estructuras e interfaces.

O, por ejemplo, acceder a la cantidad de días que posee un mes a partir de su nombre:

```
int diasMarzo = Mes["Marzo"];
```

Un diccionario deberá relacionar la información de la clave con la de los valores. Además, deberá hacerlo de un modo eficiente para su consulta. Aunque esto **no es** parte de la definición formal de lo que es un diccionario, usualmente será así, ya que de otro modo sería muy poco práctico.

Un diccionario en C# podrá poseer como tipo de valor de la **clave cualquier tipo** de dato, a la vez que también podrá ser de **cualquier tipo el valor** retornado. Es decir que podremos:

- Acceder a un string por medio de un string.
- Acceder a un número entero por medio de un número objeto X.
- Acceder a un objeto X por medio de un objeto Y, etc.

El diccionario Hashtable

La librería **BCL** provee una clase llamada **Hashtable**, que ofrece un diccionario optimizado para encontrar rápidamente un valor a partir de una clave.

La idea general en una tabla de tipo *hash* es la de acceder directamente a un elemento en una tabla sin tener que buscarlo recorriendo ésta. ¿Y cómo se logra esto? Por medio de una función **hash**, que toma la clave como entrada y provee un índice numérico como salida, que es donde se almacena el valor en la lista.
Una función **hash** trivial podría ser sumar los códigos ASCII del string por almacenar en módulo **n** (siendo **n** la cantidad total de elementos de la tabla). Claro que dos strings distintos podrían producir el mismo índice, lo cual implicaría una colisión. De todos modos, si la función **hash** se encuentra bien pensada, la cantidad de colisiones debería mantenerse en un número bajo.

LA INTERFAZ IDICTIONARY

La clase **Hashtable** posee la interfaz **IDictionary** que conviene utilizar si deseamos implementar un diccionario en una clase nuestra. Las propiedades y los métodos que define son menos de una decena, y nos permitirá integrar mucho mejor nuestra clase a la librería BCL.

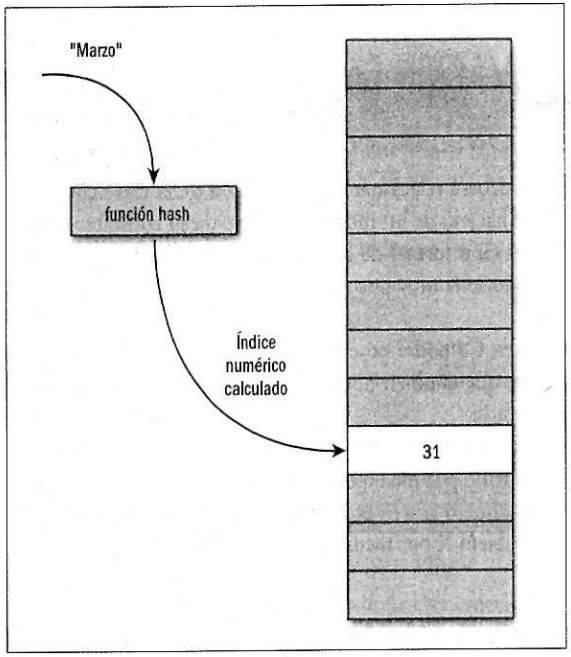


***Figura 14.** Función hash en acción.*

Cuando se produce una colisión, es decir, cuando dos entradas distintas a la función de hash producen la misma salida, no se debe devolver ningún error; la clase que implemente el algoritmo deberá tomar las acciones necesarias para poder almacenar el valor en la tabla.
Muchas veces, de cada elemento de la tabla se relaciona otra lista que contiene los elementos en colisión; claro que la búsqueda para estos elementos incurrirá en más pasos, pero nunca debería ser tan costoso como buscar por la tabla entera.

Cuando la cantidad de colisiones es elevada, podemos estar ante dos escenarios:

1. La función de hash es mala para el conjunto de datos entrantes.
2. El tamaño de la tabla es muy pequeño.

Respecto a esto último, cabe destacar que el tamaño de la tabla de hash no debería ser un número suficientemente bajo para albergar a todos los elementos, sino que se suele recomendar que sea el número primo mayor al doble de la cantidad de elementos máximos que podría poseer la tabla (sí, se desperdiciará espacio en virtud de velocidad de acceso a los elementos). De todos modos, esta recomendación se encuentra íntimamente relacionada con la función hash que utilice la implementación del algoritmo.

Haciendo uso de la clase Hashtable

La clase **Hashtable** posee diversos métodos; implementa las interfaces **ICollection**, **Ienumerable** y **IDictionary**. Por eso, algunos métodos de esta clase ya los conocemos.

```
Hashtable ht = new Hashtable(31);
ht.Add("Enero", 31);
ht.Add("Febrero", 28);
ht.Add("Marzo", 31);
ht.Add("Abril", 30);
ht.Add("Junio", 30);
ht.Add("Julio", 31);
ht.Add("Agosto", 31);
ht.Add("Septiembre", 30);
ht.Add("Octubre", 31);
ht.Add("Noviembre", 30);
ht.Add("Diciembre", 31);
Console.WriteLine("Marzo posee {0} días", ht["Marzo"]);
Console.WriteLine("Septiembre posee {0} días", ht["Septiembre"]);
Console.WriteLine("Diciembre posee {0} días", ht["Diciembre"]);
```

Analicemos el código anterior:

- El constructor se encuentra sobrecargado una decena de veces. En este caso hemos empleado aquel que recibe el tamaño total de la tabla como parámetro.
- El método nos permite ingresar en el **Hashtable** una clave (primer parámetro) y un valor (segundo parámetro). Cada parámetro puede ser de cualquier tipo, ya que son el tipo **object** y todos los tipos de datos descienden de él.
- Para obtener el valor a partir de una clave se usa un indexador implementado en la clase. Colocamos la clave entre corchetes y obtenemos el valor como retorno.

RESUMEN

No existe programa fuera de lo trivial que no haga uso de colecciones de algún tipo. Desde los clásicos arrays potenciados por la librería BCL hasta la gran variedad de tipos de colecciones que nos ofrece ésta. El problema ahora no es cómo implementar determinado tipo de colección por nosotros mismos, sino qué clase ya implementada elegir ante tantas opciones que nos ofrece esta plataforma. Un problema muy lindo, por cierto.

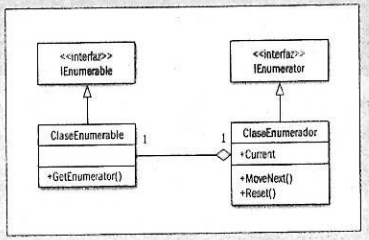
ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Cuál es el índice del primer elemento de un array?

2 ¿Qué sucede si intentamos acceder al elemento n+1 de un array de n elementos?

3 ¿Por qué motivo un array posee métodos y propiedades?



4 ¿Puede una clase implementar simultánemanente las interfaces IEnumerable e ICollection de modo?

EJERCICIOS PRÁCTICOS

✔ Crear un programa que permita:

a) Ingresar el nombre y el teléfono de una o más personas (almacenar estos datos en un objeto de tipo Hashtable).

b) Consultar el teléfono de una persona por medio del nombre.

El programa comienza solicitando el nombre de la primera persona, luego solicita el teléfono, y así continuamente, hasta que ingresa como nombre un texto vacío. Luego el programa esperará por el nombre de la persona, tras lo cual escribirá en pantalla su teléfono o la palabra "no se encuentra en registros" si éste no está en la tabla (cuando ingresa como nombre de consulta un texto vacío, el programa debe finalizar).

Capítulo 6

Delegados y eventos

Los delegados y los eventos son muy utilizados en la programación Windows. El usuario puede interactuar con ellos generando ocurrencias sobre los controles de las ventanas, y el modo más práctico de manejarlos es con funciones específicas que tratan dichas ocurrencias (eventos). C# ofrece un mecanismo simple que cumple con este objetivo y que nos servirá para muchísimos otros fines.

DELEGADOS

En ocasiones es necesario algún mecanismo que le permita al programador especificar una función o un método que será invocado si ocurre un determinado suceso.

Un ejemplo clásico es la implementación de un **timer**. Un timer es una especie de reloj despertador, sólo que, en lugar de especificarle una hora absoluta, le indicamos una cantidad de tiempo que debe transcurrir para que nos alerte.

Luego, para saber desde una clase que utilice el timer cuándo ha transcurrido el tiempo especificado, ¿qué podríamos hacer? ¿Podríamos estar consultando constantemente el objeto **timer**, preguntando si el contador ya ha llegado al número propuesto? ¡Sería muy poco eficiente!, además de poco preciso. Lo ideal sería que el objeto **timer** nos llamara a nosotros. Es decir que invertiríamos los papeles por un momento: en lugar de ser nosotros quienes invocáramos los métodos de los objetos instanciados, dicho objeto sería quien llamara a algún método de una de nuestras clases.

Pero entonces ¿cómo podría el objeto **timer** informarnos de la ocurrencia del suceso: **¡ya-transcurrió-el-tiempo-especificado!**?

De nada servirá colocar el código que necesitemos utilizar dentro de nuestra clase que implementa el timer, porque de este modo será muy poco reutilizable. Lo que aquí necesitamos es una solución general y elegante.

Históricamente los lenguajes ofrecieron diversas maneras de lidiar con esta situación. En lenguaje C, por ejemplo, es posible especificar una función **callback** que será invocada cuando sea preciso.

callback es, en realidad, una función convencional que posee un prototipo especificado (coincide en tipo de dato retornado y en cantidad y tipo de parámetros). Luego, un puntero a esta función es pasado como parámetro a la librería que lo requiera. Cuando suceda el evento en cuestión, la función **callback** será invocada.

¿QUÉ ES UN PUNTERO?

Un puntero es una variable que posee como valor una dirección de memoria. Un puntero a una función es una variable que contiene la dirección de memoria donde esta función se encuentra definida. Los punteros son populares en lenguajes como **C** y **C++**. **C#** permite emplearlos en contextos no seguros, pero casi no se utilizan. Ya veremos un poco más en el **Capítulo 10**.

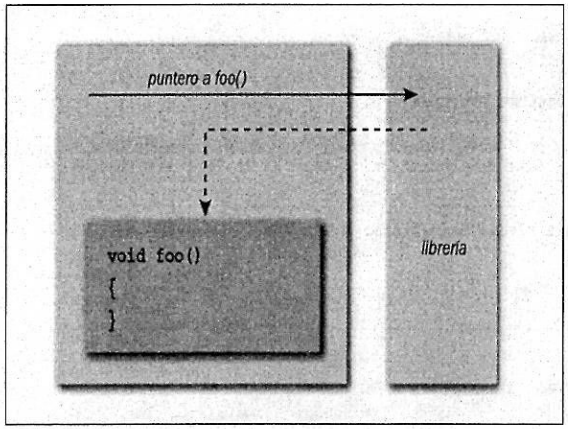


***Figura 1.** Una función callback del lenguaje C.*

El lenguaje C++, además de heredar el mecanismo **callback** del lenguaje C, permite utilizar un mecanismo que ya conocemos. Este ingenio no es ni más ni menos que la aplicación de los métodos virtuales. Recordemos que los métodos virtuales fueron ideados con el fin de separar la declaración de una interfaz con su implementación. Bueno, éste es uno de los casos en los cuales es necesario que una función sea declarada en un sitio (por ejemplo, una librería) pero sea implementada en otro (por ejemplo, nuestro sistema que hace uso de la librería).

El único problema de los métodos virtuales es que requieren estar dentro de una subclase de la clase en la cual se declara el método en cuestión. En muchas ocasiones esto es un problema e incide directamente en el diseño de nuestro sistema. Veamos nuestro ejemplo del timer implementado con métodos virtuales:

```
class Timer
{
    public void Start()
    {
        // Evitamos lidiar con el conteo del tiempo,
        // simplemente invocamos el método Tick
        Tick();
    }
    protected virtual void Tick()
    {
    }
}
```

```
class Clase1 : Timer
{
    protected override void Tick()
    {
        Console.WriteLine("Ejecución de Tick de Clase1");
    }
    static void Main(string[] args)
    {
        Clase1 obj = new Clase1();
        obj.Start();
    }
}
```

Podríamos suponer que la clase **Timer** forma parte de alguna librería, y que ni siquiera podríamos acceder a su código fuente ni modificarla en ningún aspecto. Sin embargo, declara un método virtual que es invocado cada vez que se cumple el intervalo de tiempo especificado (aunque en nuestro ejemplo no se realice un conteo, para mantener el código breve y sencillo). Entonces, para poder "**colgar**" código de este método, debemos sobrecargarlo en una subclase de **Timer**, como ya hemos hecho en capítulos anteriores.

El sistema funciona a la perfección; en el momento en que invocamos el método **Start**, éste invoca el método **Tick** y, como hemos sobrecargado, se ejecutará la versión de la clase **Clase1** y no la de **Timer**.

Como hemos mencionado anteriormente, este método suele emplearse en lenguajes orientados a objetos como C++, e incluso en C#. Pero este último va un paso más allá; ¿por qué no unir la idea general de las funciones **callback** con la practicidad de los métodos virtuales? Sería bueno poder especificar el método que será invocado en cualquier clase de nuestro sistema (sin que ésta deba ser descendiente de ninguna clase en particular), e implementar allí el código requerido. Dicho de otro

FUNCIONES CALLBACK EN C++ Y C#

El lenguaje **C++**, como sucesor del lenguaje **C** -que de algún modo lo es-, permite utilizar el mecanismo de funciones **callback**. En **C#**, en cambio, esto es imposible, pues aquí toda función debe estar definida dentro de una clase, algo que no exige **C++** (lenguaje que muchas veces es criticado por esta característica y es tildado de híbrido).

modo, queremos hacer algo parecido al ejemplo que acabamos de analizar, pero sin que **Clase1** deba ser subclase de **Timer**. Y aquí entran en acción los delegados.

¿Qué es un delegado?

Los delegados son clases especiales, y sus objetos son básicamente contenedores de referencias a métodos de otros objetos.

De esta manera, pueden ser utilizados para almacenar una referencia al método que deseamos invocar, como si fuese una función **callback** convencional pero con una flexibilidad mucho mayor, ya que el método puede encontrarse en cualquier clase; y podríamos indicar que cuando ocurra el suceso no se invoque un solo método, sino todos los que deseemos (**multicast**).

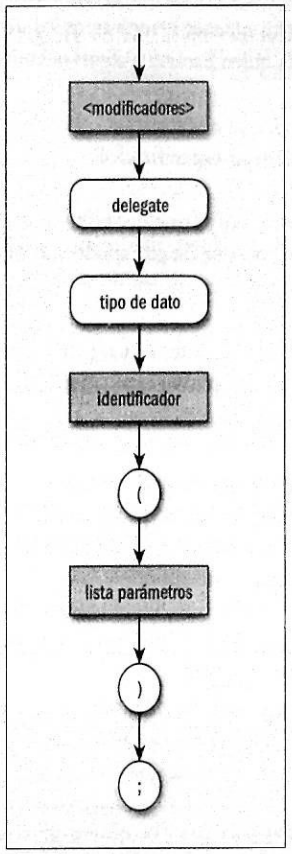


Figura 2. *Diagrama de síntaxis para la declaración de un delegado.*

Ahora veamos un ejemplo sencillo en el cual utilizamos un delegado:

```
public delegate void TickHandler();
class Clase1
{
    public Clase1()
    {
        TickHandler obj = (TickHandler)
            Delegate.CreateDelegate(typeof(TickHandler),
                this, "EsteEsElMetodoTick");
        obj(); // Aquí se invoca el método EsteEsElLmetodoTick()
    }
    public void EsteEsElMetodoTick()
    {
        Console.WriteLine("Ejecución de EsteEsElMetodoTick de Clase1");
    }
    static void Main(string[] args)
    {
        Clase1 obj = new Clase1();
    }
}
```

Como se puede apreciar en el listado anterior, primero declaramos el delegado. Luego, creamos un objeto del tipo del delegado que hemos declarado (el delegado es, al fin y al cabo, una clase) por medio de un método estático de la clase **Delegate**. La versión del método que utilizamos recibirá tres parámetros:

1. El tipo de la clase **Delegate** que hemos declarado.
2. El objeto en el cual reside el método para invocar.
3. El nombre, en string, del método para invocar (del cual luego se obtendrá la referencia que corresponda).

DELEGADOS POPULARES

El mecanismo de delegados y eventos es realmente útil. Es muy utilizado por la librería **BCL** para diversos fines. Uno de ellos, tal vez el más popular, es el de los eventos de los controles de los formularios, que comenzaremos a analizar en los siguientes capítulos.

Una vez creado el delegado, podremos utilizarlo con la finalidad de invocar el método que le hemos especificado.

Para esto simplemente debemos indicar el nombre de su instancia y su lista de parámetros entre paréntesis (en nuestro caso la lista es vacía, debido a que nuestro método no posee ningún parámetro).

La forma de invocación del método contenido en el delegado que hemos utilizado puede parecer un poco extraña a nuestra vista: el lenguaje entiende, aquí, que debe invocar el método contenido en el delegado.

El delegado es una clase

Como ya hemos mencionado, el delegado es una clase especial. Cuando declaramos un delegado, en realidad estamos declarando una clase descendiente de **System.MulticastDelegate**, sólo que lo hacemos con una sintaxis distinta. Lo que hará C# será **convertir** esta declaración en lo siguiente:

```
<modificadores> class <identificador> : System.MulticastDelegate
{
    // Declaración de algunas variables específicas
    // ...
    // Declaración de algunos métodos específicos
    // ...
    public <identificador>(object o, int punteroAMétodo) {}
    public virtual <tipo de dato> Invoke(<lista de parámetros>) {}
}
```

Pero nosotros no debemos complicarnos la vida con esta sintaxis. De hecho, la clase declara algunas variables y métodos más que ni siquiera es menester mencionar aquí.

Simplemente es bueno tener en cuenta qué es lo que el lenguaje entiende realmente y cuál es la naturaleza de este tipo de objeto nuevo que estamos utilizando.

EVENTOS

Un evento es un tipo especial de propiedad y está muy relacionado con los delegados, ya que se utiliza para controlar el modo en que nuestro sistema hace uso de ellos.

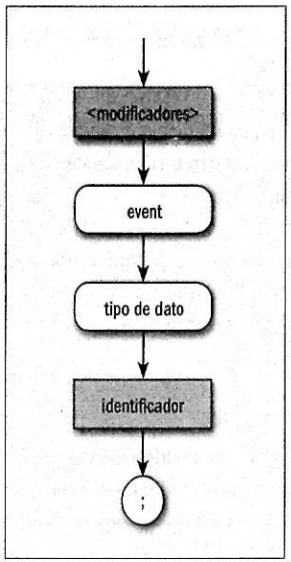


Figura 3. *Diagrama de declaración de un evento.*

Ahora volvamos a la implementación de nuestro timer. Haciendo uso de delegados y eventos, obtendremos una solución ideal a nuestras necesidades:

```
public delegate void TickHandler();
public class Timer
{
     public void Start()
     {
          // Invoco el evento Tick
          if (Tick != null)
               Tick();
     }
     public event TickHandler Tick = null;
}
class Clase1
{
     Timer m_timer;
     public Clase1()
     {
          // Creo el objeto timer
          m_timer = new Timer();
          // Establezco el delegado
          m_timer.Tick += new TickHandler(this.EsteEsElMetodoTick);
          // Inicio el timer
          m_timer.Start();
     }
     public void EsteEsElMetodoTick()
     {
          Console.WriteLine("Ejecución de EsteEsElMetodoTick de Clase1");
     }
     static void Main(string[] args)
     {
          Clase1 obj = new Clase1();
     }
}
```


Analicemos el código anterior:

1. Declaramos un delegado, indicando que éste deberá almacenar la referencia a un método sin parámetros.
2. Nuestra clase **Timer** poseerá un evento inicializado en **null** que gestionará el acceso al delegado **TickHandler**.
3. La clase **Clase1** crea un objeto del tipo **Timer**.
4. Hacemos uso del evento para relacionar el delegado con el método del objeto que deseamos invocar. Note que para esto utilizamos el operador **+=**. Podríamos seguir agregando métodos para ser invocados por el delegado mediante este operador, y podríamos indicar lo contrario por medio del operador **-=**.
5. Finalmente, cuando invocamos el método **Start**, éste hace uso del evento para invocar el método contenido en el delegado.

Entonces hemos solucionado el problema que teníamos. Logramos indicarle a un objeto de nuestro sistema que cuando se produzca un determinado suceso invoque

INTERRUPCIÓN DE LA EJECUCIÓN EN CADENA

Es posible que se interrumpa la invocación en cadena de los métodos que un delegado tiene referenciados si ocurre una excepción en alguno de ellos y ésta es arrojada hacia el método superior.

a un método arbitrario definido por nosotros y situado en una de nuestras clases, sin que ésta deba poseer una relación específica de herencia con ninguna otra.

Cuando se presiona un botón en uno de nuestros formularios, podremos indicarle al framework que invoque el método que queramos para su tratamiento. De hecho, podremos tener una clase que represente el formulario y especificar en ella los métodos que deben ser invocados para todos los botones –y demás controles– que posea el formulario. ¡Qué poco práctico sería aquí tener que crear una subclase específica por cada botón para tratar alguno de sus eventos!

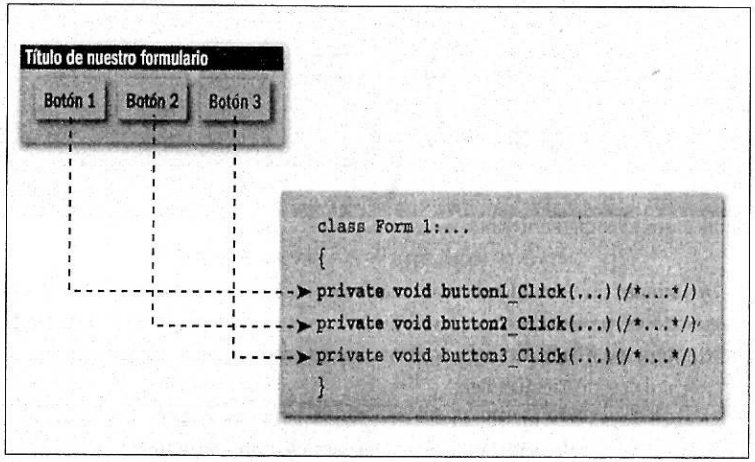


Figura 4. *Uso de eventos y delegados en* ***GUIS.***

Invocar múltiples métodos

Ya dijimos que los delegados pueden no sólo invocar un método, sino todos los que queramos. Delegados del tipo **MulticastDelegate** almacenan listas enlazadas de referencias a métodos, por lo que podemos solicitar la invocación de todos los métodos albergados en cadena, uno detrás de otro (pero sin suponer un orden particular).

¿QUÉ ES UNA EXCEPCIÓN?

Una excepción es un recurso del lenguaje que permite alterar el flujo de ejecución normal de un programa. Usualmente son utilizadas para tratar los errores que se pueden producir en tiempo de ejecución en nuestras aplicaciones. Más adelante veremos cómo hacer uso de ellas. Aprenderemos más acerca de excepciones en el **Capítulo 9**.

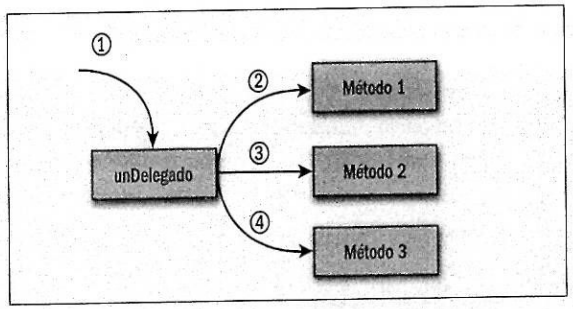


Figura 5. *Ejecución de métodos en cadena.*

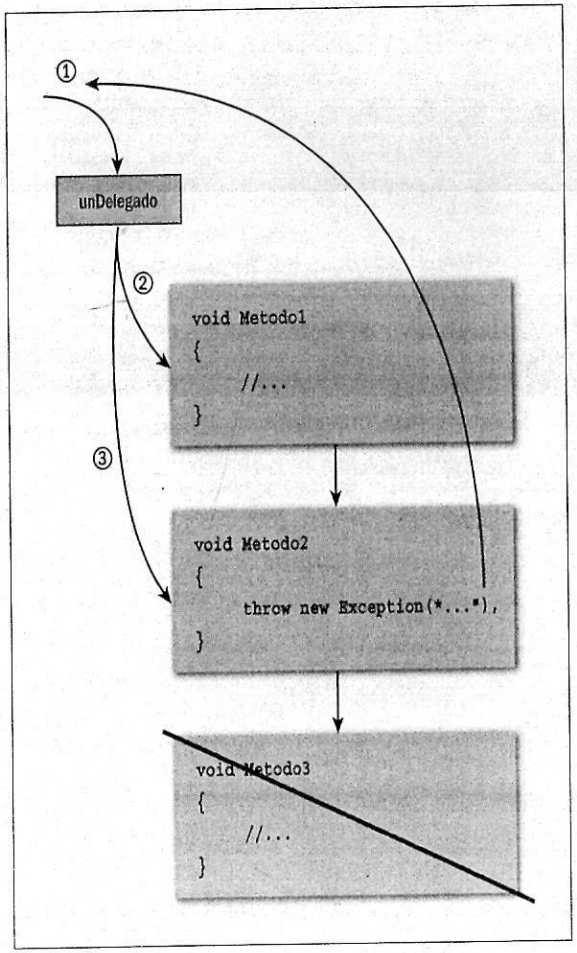


Figura 6. *El método* ***Metodo3*** *nunca será invocado.*

Veamos cómo hacer esto basándonos en un ejemplo trivial:

```
class Clase1
{
    Timer m_timer;
    public Clase1()
    {
        // Creo el objeto timer
        m_timer = new Timer();
        // Establezco el delegado
        m_timer.Tick += new TickHandler(this.EsteEsElMetodoTick1);
        m_timer.Tick += new TickHandler(this.EsteEsElMetodoTick2);
        m_timer.Tick += new TickHandler(this.EsteEsElMetodoTick3);
        // Inicio el timer
        m_timer.Start();
    }
    public void EsteEsElMetodoTick1()
    {
        Console.WriteLine("Ejecución de EsteEsElMetodoTick1
            de Clase1");
    }
    public void EsteEsElMetodoTick2()
    {
        Console.WriteLine("Ejecución de EsteEsElMetodoTick2
            de Clase1");
    }
    public void EsteEsElMetodoTick3()
    {
        Console.WriteLine("Ejecución de EsteEsElMetodoTick3
            de Clase1");
    }
    static void Main(string[] args)
    {
        Clase1 obj = new Clase1();
    }
}
```

Es totalmente obvio que esto resultaría muchísimo más práctico si los métodos por invocar fuesen de distintas clases.

LA CLASE SYSTEM.TIMERS.TIMER

La librería **BCL** nos ofrece una clase **Timer** que es ideal para utilizar en nuestras aplicaciones cuando éstas deben realizar una determinada tarea cada cierto período de tiempo. Esta clase hace uso de delegados y eventos (como no podía ser de otro modo). Veremos aquí cómo podemos utilizarla:

```
using System;
using System.Timers;
namespace UsoDeTimer
{
    class Clase1
    {
        Timer _timer = new Timer();
        Clase1()
        {
            // Indico que el intervalo sea de 1000 ms
            _timer.Interval = 1000;
            // Especifico el método que será invocado por el delegado
            _timer.Elapsed += new ElapsedEventHandler(this.Tick);
            // Le doy comienzo al timer
            _timer.Start();
        }
        public void Tick(object sender, ElapsedEventArgs e)
        {
            // Envío a pantalla un mensaje cada vez que se
            // invoca el método Tick
            Console.WriteLine("Tick");
        }
        static void Main(string[] args)
        {
            // Creo un objeto del tipo Clase1
            Clase1 c = new Clase1();
            // Solicito el ingreso de un string, para que el
            // programa no termine
            string str = Console.ReadLine();
        }
    }
}
```

Como se aprecia en el listado anterior, no existen grandes diferencias respecto al código que veníamos escribiendo. Tal vez deberíamos notar que el método para invocar posee dos argumentos: el primero es el objeto que utiliza el evento para invocar nuestro método (es decir, el objeto **_timer**), y el segundo posee información relacionada con el evento que se produce, en este caso, el transcurso de cierto tiempo.

Para conocer cuál será el prototipo del método (tipo de dato retornado y parámetros) que debemos definir, debemos consultar la información de referencia oficial que acompaña al **framework .NET SDK**.

CASO DE ESTUDIO

Construir un pequeño programa que conste de tres clases (**Clase1**, **Clase2** y **Clase3**). Un objeto de **Clase3** deberá contener un objeto de **Clase2**, el cual deberá contener un objeto de **Clase1**.

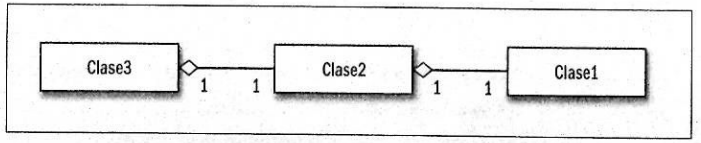


***Figura 7.** Relación de contención entre objetos de la clase.*

Declarar, además, un delegado que permita almacenar referencias a métodos con un parámetro entero.

Clase1 y **Clase2** deben poseer un evento relacionado con el delegado declarado. Además, deben contener un método llamado **Start** que cuando sea invocado en **Clase1** dispare su evento, lo cual producirá la invocación de un método de **Clase2**, que dispara su propio evento, y esto producirá finalmente la invocación de dos eventos distintos de la clase **Clase3**.

DELEGADOS EN JAVA

Java no soporta delegados. Para realizar implementaciones similares a las vistas, suelen utilizarse interfaces de un solo método que la clase en cuestión debe sobrecargar.

El siguiente diagrama de secuencia clarifica lo mencionado:

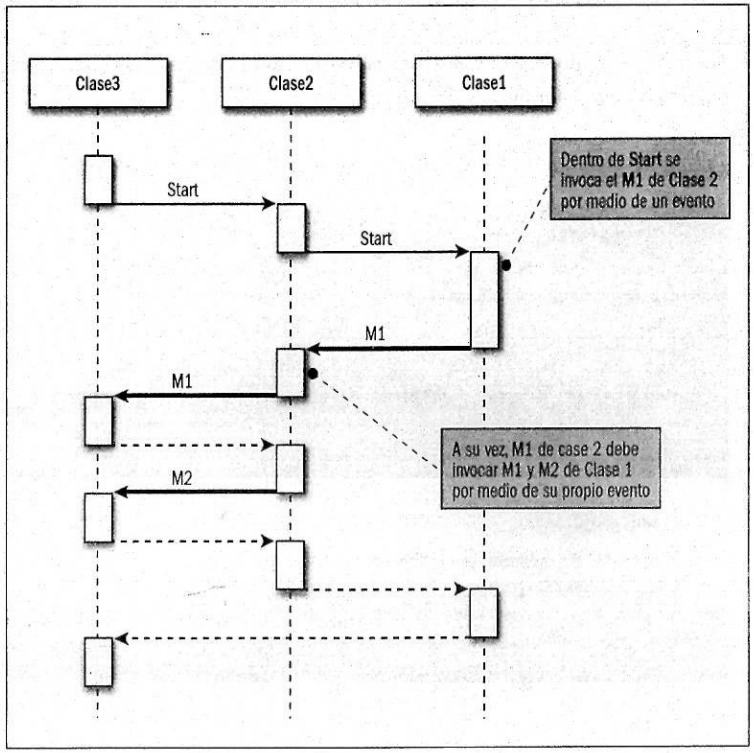


***Figura 8.** El diagrama de secuencia del ejercicio.*

Resolución

Con relación al enunciado escrito anteriormente: "**Declarar, además, un delegado que permita almacenar referencias a métodos con un parámetro entero**", simplemente debemos ingresar como código:

EVENTOS DE CLASES EN LA DOCUMENTACIÓN OFICIAL

En la documentación oficial de referencia que incluye Visual Studio .NET (a la cual se puede acceder en forma libre desde Internet), se identifican fácilmente los eventos que implementan las clases (así como los métodos y las propiedades), debido a que están perfectamente categorizados.

```
public delegate void MiDelegado(int num);
```

Prestar especial atención a que ahora nuestro delegado posee un parámetro entero que nos solicita el enunciado.

Definir las tres clases con sus propios métodos y eventos es muy sencillo:

```
public class Clase1
{
    public void Start(int num)
    {
    }
    public event MiDelegado Evento = null;
}
class Clase2
{
    Clase1 objClase1;
    public event MiDelegado Evento = null;
    public Clase2()
    {
    }
    public void M1(int num)
    {
    }
    public void Start(int num)
    {
    }
}
class Clase3
{
```

III ¿QUÉ SUCEDE SI INVOCAMOS UN EVENTO NULO?

Inicialmente, los eventos poseen asociado el valor nulo. Si aun así lo utilizamos para invocar los delegados asociados, se producirá una excepción del tipo **System.NullReferenceException**.


```
    Clase2 objClase2;
    public Clase3()
    {
    }
    public void M1(int num)
    {
    }
    public void M2(int num)
    {
    }
    public void Start(int num)
    {
    }
}
```

Ahora deberemos relacionar la invocación de métodos y eventos, según solicita el enunciado. Completando entonces el programa:

```
using System;
namespace DelegadosYEventosEjercicio1
{
    public delegate void MiDelegado(int num);
    public class Clase1
    {
        public void Start(int num)
        {
            // Invoco el evento
            if (Evento != null)
                Evento(num);
        }
        public event MiDelegado Evento = null;
    }
    class Clase2
    {
        Clase1 objClase1;
        public event MiDelegado Evento = null;
        public Clase2()
        {
            // Creo el objeto
```

```
            objClase1 = new Clase1();
            // Establezco el delegado
            objClase1.Evento += new MiDelegado(this.M1);
        }
        public void M1(int num)
        {
            Console.WriteLine("Ejecución de M1 de Clase2.
                Número = {0}", num);
            // Invoco el evento
            if (Evento != null)
                Evento(num);
        }
        public void Start(int num)
        {
            objClase1.Start(num);
        }
    }
    class Clase3
    {

        Clase2 objClase2;
        public Clase3()
        {
            // Creo el objeto
            objClase2 = new Clase2();
            // Establezco el delegado
            objClase2.Evento += new MiDelegado(this.M1);
            objClase2.Evento += new MiDelegado(this.M2);
        }
        public void M1(int num)
        {
            Console.WriteLine("Ejecución de M1 de Clase3.
                Número = {0}", num);
        }
        public void M2(int num)
        {
            Console.WriteLine("Ejecución de M2 de Clase3.
                Número = {0}", num);
        }
        public void Start(int num)
```



```
        {
            objClase2.Start(num);
        }
        static void Main(string[] args)
        {
            Clase3 obj = new Clase3();
            // Invoco Start con un número cualquiera
            obj.Start(5);
        }
    }
}
```

RESUMEN

Finalmente hemos visto los conceptos más importantes del lenguaje que nos permitirán sumergirnos de lleno en la programación de aplicaciones de ventana.

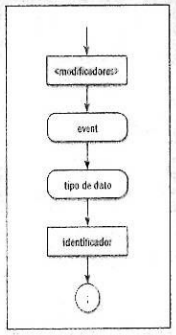
En el próximo capítulo veremos cómo, por medio de la librería BCL, podremos crear aplicaciones con botones, cuadros de texto, listas y muchísimos controles más. Aprenderemos, entonces, que los delegados y los eventos forman parte de los recursos más populares que utilizaremos para colgar código de los sucesos que pueden producirse en los controles y en el formulario.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

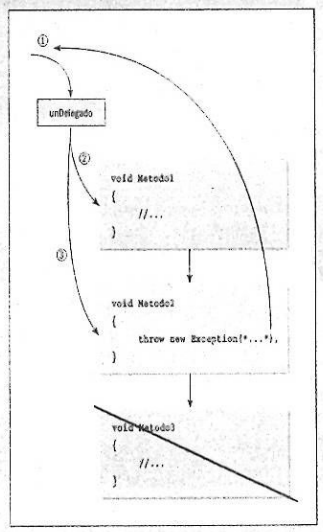
1 ¿Qué es realmente un delegado?

2 ¿Para qué se utilizan usualmente los delegados y los eventos?



3 ¿Qué diferencia sustancial existe entre invocar un método e invocar un método mediante un evento?

4 ¿Qué ventajas conlleva utilizar delegados y eventos respecto a métodos virtuales?



5 ¿En qué namespaces de la librería BCL son más populares los delegados y eventos?

Capítulo 7

Aplicaciones con Windows.Forms

Ha llegado el momento de introducirnos en la creación de aplicaciones con ventanas y controles. Por eso, en este capítulo daremos los pasos básicos para crear una aplicación con interfaz de usuario. Presentaremos los controles básicos de la librería y veremos cuáles son sus métodos y propiedades básicas.

SERVICIO DE ATENCIÓN AL LECTOR: lectores@mpediciones.com

ARQUITECTURA DE UNA APLICACIÓN WINDOWS

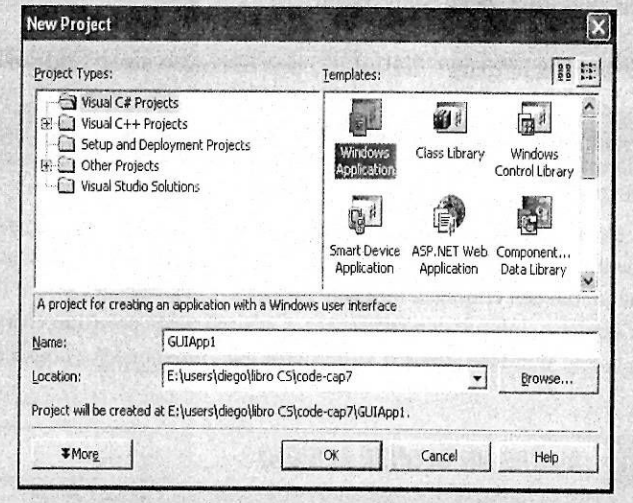
Toda aplicación para **Windows** que requiera interacción con el usuario y que se precie de tal cuenta con una interfaz gráfica (**GUI**). En el pasado han quedado aquellos sistemas en los cuales solamente podíamos utilizar, por medio de la línea de comandos, clásicos del viejo y vetusto **DOS**.

C# permite crear aplicaciones de consola y con interfaz de usuario. Desde el comienzo del libro hasta aquí sólo hemos tratado el primer tipo de aplicaciones. Finalmente, ha llegado el momento de adentrarnos en la construcción de aplicaciones más bonitas y, desde varios puntos de vista, más interesantes.

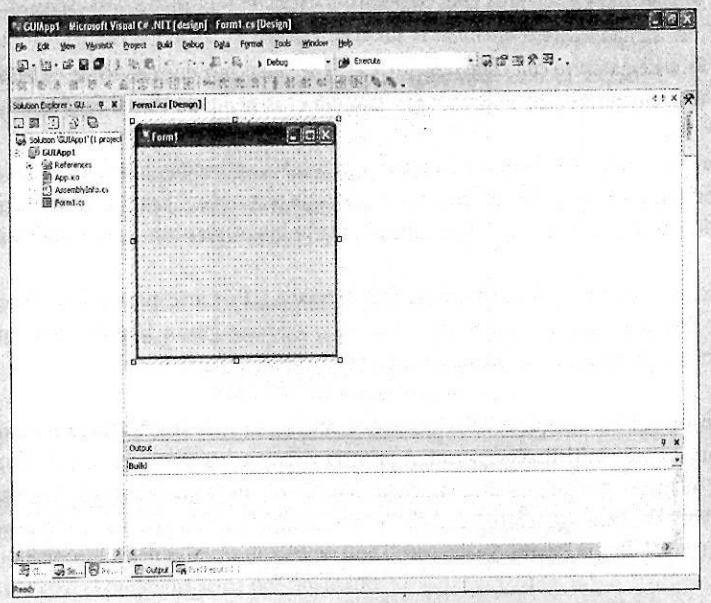
Veamos, paso a paso, cómo crear una sencilla aplicación con interfaz de usuario utilizando **Visual Studio .NET 2003**.

■ Crear una aplicación con interfaz de usuario PASO A PASO

1 En primer lugar deberemos comenzar, como siempre, accediendo a la opción de nuevo proyecto (**New Project**).



2 Especificaremos ubicación y nombre del proyecto, pero en esta oportunidad no seleccionaremos **Console Application**, sino que elegiremos **Windows Application**, y luego presionaremos el botón **Ok**.



3 Si todo ha ido bien, deberíamos ver en nuestra pantalla algo similar a lo que expresa la figura anterior.

Ahora bien, en la pestaña **Solution Explorer** encontraremos todos los archivos que pertenecen al proyecto o están relacionados con él; algunos nos resultan conocidos y otros no. Veamos:

- **Carpeta References**: dentro de esta carpeta se listarán los distintos **assemblies** que utiliza nuestro proyecto. Recordemos que un **assembly** es un componente de software que en su interior posee código ejecutable y/o recursos (los **assemblies**, que estarán relacionados con nuestros proyectos, se encontrarán generalmente en archivos de tipo **dll**). Por el momento, entre ellos podremos encontrar:

System	**System.Windows.Forms**
System.Data	**System.XML**
System.Drawing	

- **App.ico**: el icono que representa nuestra aplicación. Puede ser modificado mediante el editor integrado que posee **Visual Studio** (aunque es recomendable trabajar con editores externos si es que se desea construir iconos con más de 16 colores).

- **AssemblyInfo.cs**: archivo con código fuente C#, posee atributos del **assembly** de nuestra aplicación.
- **Form1.cs**: archivo con código fuente C#. Posee una clase que desciende de **System.Windows.Forms.Form** y código generado por el asistente de **Visual Studio**.

La clase **Form1** representa un formulario en nuestra aplicación. Los formularios son contenedores de controles (botones, cuadros de texto, etc.). Éstos serán útiles para construir una interfaz que le permita al usuario interactuar con nuestra aplicación.

Existen muchos tipos de controles, el **framework 1.1** incluye más de 50 y existe en la Red una oferta de muchísimos otros para todos los gustos (algunos gratuitos y otros comerciales), que podremos incorporar a nuestras aplicaciones.

Entre los métodos que el asistente escribe en nuestra clase **Formulario** se encuentra uno llamado **InitializeComponent**. A primera vista, este método no estará visible ante nuestros ojos, pues se encuentra dentro de un bloque **region** llamado **"Windows Form Designer generated code"**. Si expandimos este bloque, encontraremos el método buscado dentro de él.

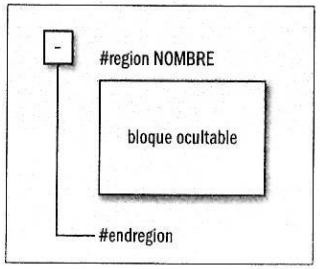


Figura 1. *Bloque expandido.*

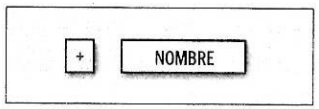


Figura 2. *Bloque colapsado.*

Dentro del método **InitializeComponent**, el diseñador de **Visual Studio .NET** colocará las propiedades de los controles a las cuales les hayamos especificado valores distintos de los predeterminados.

¿QUÉ ES UN ATRIBUTO?

Un atributo brinda información acerca de los datos que describen nuestra aplicación. En el **Capítulo 10** hablaremos más acerca de ellos: veremos cuáles son los atributos predefinidos y cómo declarar los nuestros propios.

Si compilamos y ejecutamos la aplicación, presionando la tecla **F5** o accediendo al menú **Debug**, opción **Start**, veremos en pantalla la siguiente ventana:

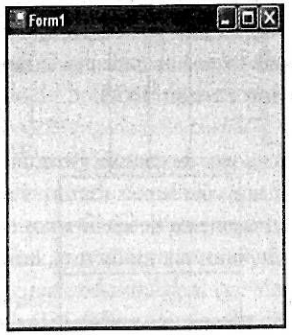


Figura 3. *La aplicación en modo de ejecución.*

Es importante notar que, a diferencia de las aplicaciones de tipo consola que antes creamos, ahora la aplicación no termina inmediatamente. Es decir que no existen –visiblemente– ciertas líneas de código que deben ser ejecutadas secuencialmente, tras lo cual el programa se da por finalizado.
Ahora, el programa se ejecuta hasta que cerremos la ventana principal o lo expresemos explícitamente mediante código.

De hecho, es interesante analizar el código de nuestro método estático **Main**:

```
Application.Run(new Form1());
```

El método estático **Run** de la clase **Application** ejecutará un bucle de mensajes para nuestra aplicación. Pero ¿qué es un bucle de mensajes? En **Windows** los procesos se comunican con el sistema mediante mensajes. Estos mensajes poseen un código y, en ocasiones, información relacionada. Cada aplicación posee una cola de mensajes en la cual **Windows** deposita los mensajes que corresponden a la aplicación.

¿QUÉ ES UN BLOQUE REGION?

Los bloques **region** nos permiten mejorar la legibilidad del código de nuestro programa. Están compuestos por un par de directivas dentro de las cuales todo el código escrito podrá ser expandido o colapsado. De este modo, si el listado total de una hoja de código fuente es muy extenso, con este tipo de bloques podremos reducir su longitud visible, agrupando métodos afines.

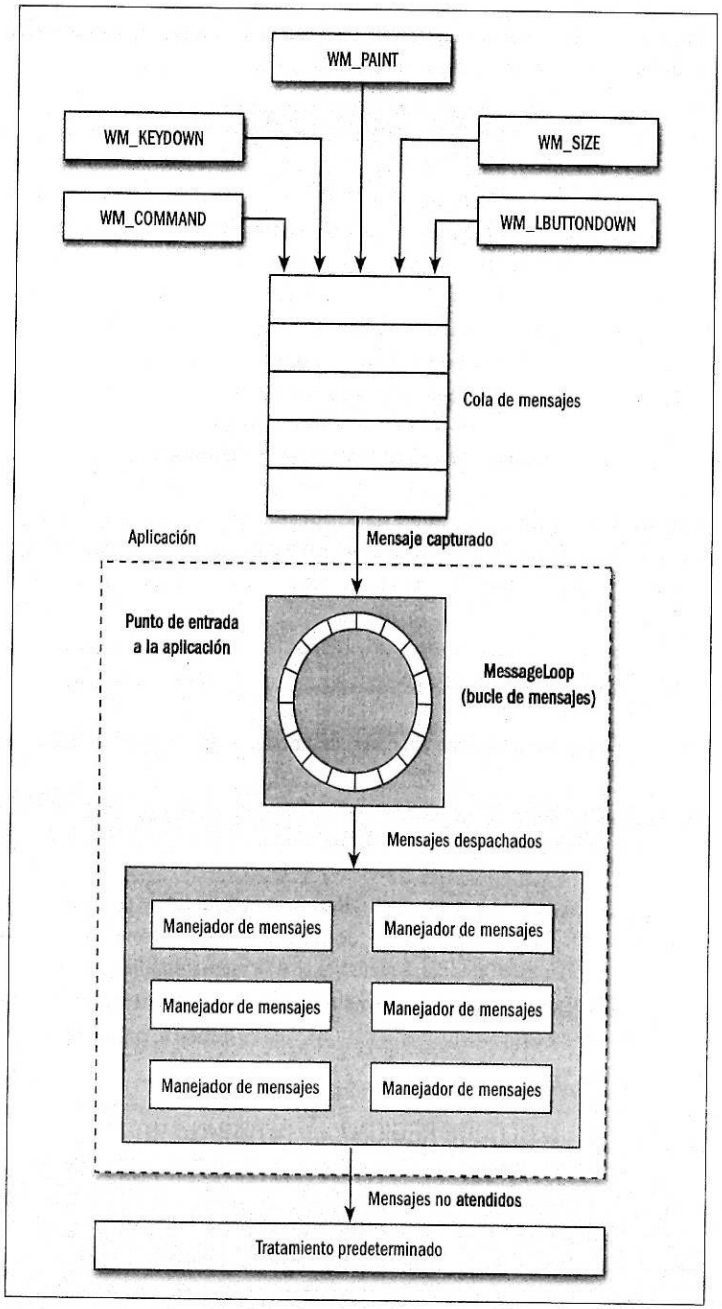


Figura 4. *La cola de mensajes.*

Por ejemplo, si el usuario presiona un botón del mouse, entonces **Windows** generará un mensaje del tipo **WM_LBUTTONDOWN**, que usualmente ingresará en la cola de mensajes de la aplicación que posea el foco en ese momento (la aplicación activa). Cada aplicación se encuentra en un bucle, esperando a que ingrese algún mensaje. Cada vez que un mensaje ingresa, la aplicación lo despacha con el manejador que corresponda. Volviendo al ejemplo del botón del mouse, nuestro programa podría poseer una función que realice una determinada tarea cuando el botón izquierdo del mouse es presionado; si no es así, el mensaje será tratado por un manejador predeterminado. En C#, todo este mecanismo de bucle de mensajes queda un tanto oculto (no sería así si programáramos en lenguaje **C**). El sistema de despacho de mensajes será implementado por el framework, y nosotros nos limitaremos a especificar métodos que respondan a eventos de un modo similar al que vimos en el capítulo anterior, cuando no sea el mismo **Visual Studio** quien escriba dicho código por nosotros.

La clase Application

Application es una clase que representa a nuestra aplicación. Posee una serie de métodos y propiedades estáticos que nos brindará una gran cantidad de información respecto a ella. Las propiedades más importantes son:

- **CompanyName**: nos devuelve información del atributo **AssemblyCompany** que especificamos para nuestro **assembly** (por predefinición, en el archivo **AssemblyInfo**).
- **ProductName**: nos devuelve información del atributo **AssemblyProduct** que especificamos para nuestro **assembly**.
- **ProductVersion**: nos devuelve información del atributo **AssemblyVersion** que especificamos para nuestro **assembly**.
- **ExecutablePath**: ruta de ejecución de la aplicación.
- **StartupPath**: ruta de arranque de la aplicación.

Y los métodos estáticos más importantes son:

- **DoEvents**: a los programadores de **Visual Basic** les resultará conocido este método. Aunque en aquel lenguaje se encontrase como una función global, aquí su propósito es el mismo: atender otros mensajes mientras nuestra aplicación se encuentra trabajando en algún proceso que demande cierto tiempo. Por ejemplo, si nuestra aplicación se encuentra dentro de un bucle extenso, es normal que un programador coloque dentro de él un **DoEvents**. De este modo, nuestra aplicación seguirá atendiendo los mensajes en cola y evitaremos que, por ejemplo, el formulario quede sin repintarse.
- **Exit**: finaliza la aplicación.
- **Run**: comienza una aplicación ejecutando un bucle de mensajes.

CONTROLES

Para modificar las propiedades de cualquier elemento bastará con seleccionarlo, presionar el botón derecho del mouse sobre él y acceder a la opción del menú contextual llamada **Properties**. Hecho esto, aparecerá la página de propiedades del control en la cual podremos observar y modificar sus propiedades, y también podremos agregar métodos que manejen sus eventos.

Además, toda modificación de propiedades se puede realizar mediante código. Lo que sucede es que, aquí, Visual Studio genera dicho código por nosotros, pero que no estemos utilizando este entorno no significa que estemos imposibilitados de trabajar con controles ni mucho menos.

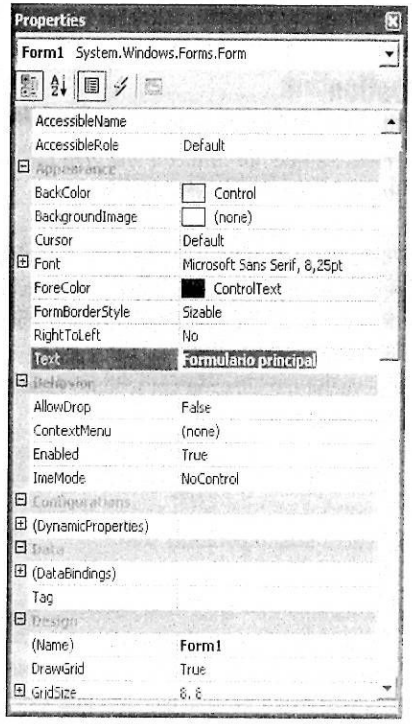


***Figura 5.** La página de propiedades.*

Es importante notar que **los controles son clases**, y cuando seleccionamos un control del **toolbox** y lo insertamos en nuestro formulario, estamos creando una instancia de dicha clase (un objeto) que **Visual Studio** automáticamente agrega a nuestro formulario con el nombre que indique la "propiedad" **(Name)**.

Existen controles que poseen representación gráfica (la mayoría) y otros que no (como **Timer**, **ImageList**, etc.):

- Los componentes sin representación gráfica ofrecen sus funcionalidades al programador, pero son invisibles en tiempo de ejecución (significa que no dibujan sobre el formulario).
  Este tipo de controles desciende de la clase **System.ComponentMode.Component**.

- Por otro lado, los controles con representación gráfica ocupan un espacio en nuestro formulario, y en tiempo de ejecución dibujan sobre su espacio algún contenido. Este tipo de controles desciende de la clase **System.Windows.Forms.Control** (que a su vez desciende de **System.ComponentMode.Component**).

En la siguiente tabla, podemos ver los controles más importantes que provee el **framework .NET** en su versión 1.1:

CONTROL	DESCRIPCIÓN
Label	Representa una etiqueta sencilla, un trozo de texto no seleccionable ni editable por el usuario.
LinkLabel	Representa una etiqueta con un enlace dentro de ella. Cuando posamos el puntero del mouse sobre ésta y hacemos un clic, automáticamente se abre el navegador predeterminado del sistema apuntando al sitio que especifica el control.
Button	Representa el clásico botón. Puede contener un texto y/o una imagen relacionada con él.
TextBox	Representa un cuadro de texto editable y seleccionable (si sus propiedades así lo determinan). Es el modo tradicional de ingresar datos por un formulario.
MainMenu	Representa el menú de una ventana que es relacionado con la propiedad Menu de un formulario. En tiempo de diseño se permite modificar las opciones y propiedades de los elementos que lo componen.
CheckBox	Representa un cuadro de verificación. Es utilizado para campos que pueden tomar dos valores (verdadero/falso, sí/no, encendido/apagado, etc.), aunque también se permite un tercer estado que es indeterminado.
RadioButoon	Representa un botón de opción que puede tomar el valor de marcado o no marcado. Usualmente se combinan varios botones de este tipo, donde su valor de verdad es mutuamente excluyente con el resto del conjunto.
GroupBox	Representa una agrupación de controles, enmarcados y con una etiqueta. Es ideal para agrupar opciones afines dentro de un formulario complejo.
PictureBox	Representa un cuadro que puede poseer una imagen en su interior.
Panel	Representa un contenedor de controles. Es práctico en muchos casos, como cuando se desea controlar la propiedad de visibilidad de muchos controles afines de modo simple.
DataGrid	Representa una grilla relacionada con un conjunto de datos. Facilita muchísimo la interacción con bases de datos.
ListBox	Representa un cuadro de lista sencillo que siempre está desplegado.
CheckedListBox	Representa un cuadro de lista sencillo donde los elementos poseen asociados cuadros de verificación.
ComboBox	Representa un control combinado de cuadro de texto y cuadro de lista.

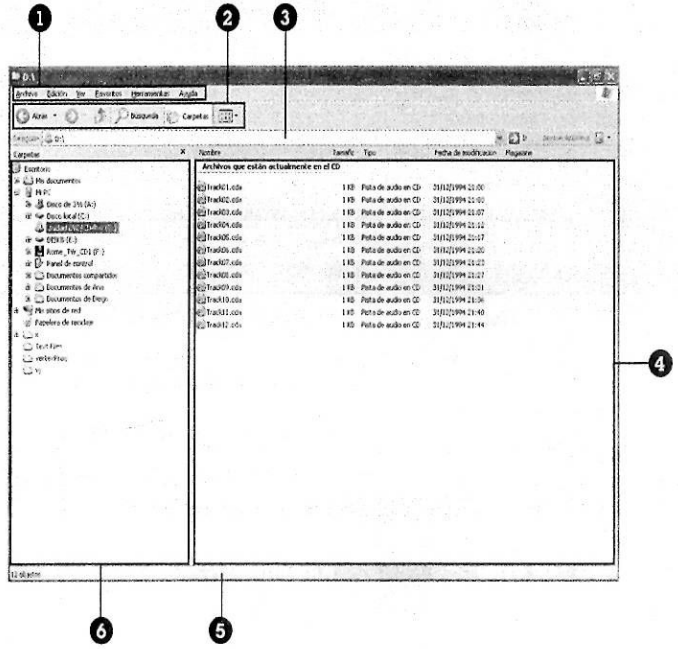
CONTROL	DESCRIPCIÓN
ListView	Representa una lista de visualización. Este control es muy flexible y permite diversas vistas. Es utilizado por muchas aplicaciones, como el explorador de archivos (cuadro derecho donde se muestran los contenidos de las carpetas).
TreeView	Representa un árbol de vistas. Este control se complementa muy bien con el ListView. Muchas aplicaciones hacen uso de él, como el mismo Visual Studio y el explorador de archivos.
TabControl	Representa un control de pestañas que es muy útil para organizar grandes cantidades de opciones dentro de una ventana pequeña.
DateTimePicker	Representa un control en el cual se ofrece un calendario para seleccionar una fecha en él cuando es desplegado.
MonthCalendar	Representa un control en el cual se ofrece un calendario para seleccionar una fecha en él. Es similar a DatePicker, excepto que aquí el calendario siempre se encuentra desplegado.
HScrollBar	Representa una barra de desplazamiento horizontal.
VScrollBar	Representa una barra de desplazamiento vertical.
Timer	Representa un timer.
Splitter	Representa un divisor. Es muy útil para otorgar al usuario la posibilidad de ajustar el tamaño de ciertos controles de modo automático (como lo que ocupa un TreeView asociado a ListView).
DomainUpDown	Representa una lista de strings que puede ser modificada secuencialmente mediante botones.
NumericUpDown	Representa una lista de números que puede ser modificada secuencialmente mediante botones.
TrackBar	Representa una barra de desplazamiento.
ProgressBar	Representa una barra de progreso.
RichTextBox	Representa un cuadro de texto con formato de texto enriquecido. De este modo es posible cambiar el color y la fuente de una parte del texto que se encuentra en su interior (a diferencia del cuadro de texto normal, donde las propiedades de tipografía y color eran generales para todo el control).
ImageList	Representa una lista de imágenes. Este control se utiliza asociado a otros que requieren imágenes (por ejemplo, TreeView, ListView, etc.). Es posible compartirlo entre varios controles.
ToolTip	Representa una pequeña etiqueta flotante que brinda una ayuda sobre algún control del formulario.
ContextMenu	Representa un menú contextual que puede ser abierto cuando el usuario realiza determinada acción con nuestra aplicación (por ejemplo, presiona el botón derecho sobre algún elemento).
Toolbar	Representa una barra de herramientas (generalmente colocada debajo del menú de la aplicación).
StatusBar	Representa una barra de estado (generalmente situada al pie del formulario).
NotifyIcon	Representa un icono que aparece en el tray (iconos que se encuentran a la derecha del reloj en la barra de tareas de Windows). Es posible asociarle un ContextMenu.
OpenFileDialog	Representa un diálogo para abrir archivos.
SaveFileDialog	Representa un diálogo para grabar archivos.
FolderBrowser	Representa un diálogo para navegar por carpetas.
FontDialog	Representa un diálogo para seleccionar tipografías.
ColorDialog	Representa un diálogo para seleccionar colores.
PrintDialog	Representa un diálogo de impresión.

***Tabla 1.** Controles que provee framework .NET 1.1.*

En la **Guía Visual 1** vemos los principales elementos que componen una ventana de Windows y cómo están identificados de acuerdo con el **framework .NET**.

Ventanas de Windows

GUÍA VISUAL 1



1. Menu.
2. Toolbar.
3. ComboBox.
4. ListView.
5. StatusBar.
6. TreeView.

DOCUMENTACIÓN OFICIAL

Recordemos que la documentación electrónica que provee **Visual Studio .NET** es ideal para utilizar como referencia cuando deseamos conocer las propiedades, métodos o eventos que posee un control. También podemos acceder a esta ayuda en línea mediante un navegador en el sitio: **http://msdn.microsoft.com/library**.

Veamos, ahora, cuáles son las propiedades comunes y más populares de los controles que poseen representación gráfica:

PROPIEDAD	DESCRIPCIÓN
BackColor	Color de fondo del control.
Dock	Permite especificar contra qué extremo del formulario se encuentra pegado el control.
Enabled	Habilita o deshabilita el control (cuando un control se encuentra deshabilitado no es posible interactuar con él).
Font	Tipografía utilizada (fuente, tamaño, color, etc.).
ForeColor	Color principal utilizado por el control.
Left	Posición izquierda del control en relación con el padre (formulario u otro control donde está contenido).
Height	Alto en pixeles.
Text	Texto asociado al control.
TextAlign	Alineación del texto asociado al control.
Top	Posición superior del control en relación con el padre (formulario u otro control donde está contenido).
Visible	Muestra u oculta el control.
Width	Ancho en pixeles.

***Tabla 2.** Propiedades de controles con representación gráfica.*

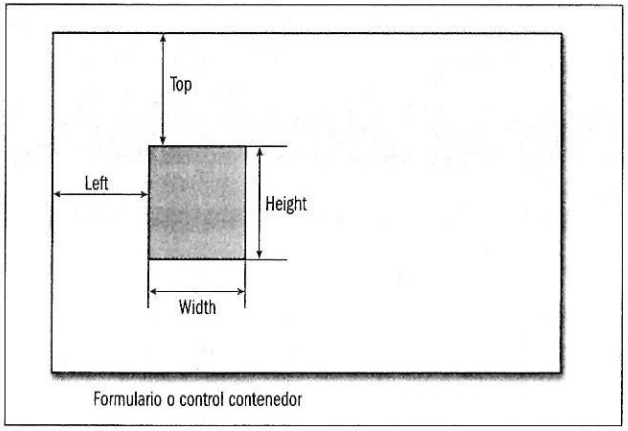


***Figura 6.** Alto, ancho y posición del control.*

PROPIEDADES, MÉTODOS Y EVENTOS

Aquí estamos citando las propiedades, métodos y eventos que consideramos más importantes (los que van a utilizarse con mayor frecuencia). Existen muchos otros que pueden consultarse en la documentación de referencia oficial que acompaña al **framework SDK** o a **Visual Studio .NET**, o incluso en el sitio **MSDN**.

Y los métodos más importantes son:

MÉTODO	DESCRIPCIÓN
Focus	Transfiere el foco al control (la entrada por teclado es dirigida al control que posea el foco).
Hide	Oculta el control (mismo efecto que fijar la propiedad Visible en false).
Show	Muestra el control (mismo efecto que fijar la propiedad Visible en true).

***Tabla 3.** Métodos de controles con representación gráfica.*

Y, finalmente, los eventos:

EVENTO	DESCRIPCIÓN
Click	Se produce cuando el usuario presiona el botón izquierdo del mouse.
DoubleClick	Se produce cuando el usuario realiza doble clic con el mouse.
GetFocus	Se produce cuando el control gana foco.
KeyDown	Se produce en el momento en que la tecla es presionada.
KeyPress	Se produce cuando se presiona una tecla sobre el control.
KeyUp	Se produce en el momento en que la tecla es liberada.
LostFocus	Se produce cuando el control pierde el foco.
MouseDown	Se produce cuando el usuario presiona un botón del mouse.
MouseMove	Se produce cuando el puntero del mouse se mueve encima del control.
MouseUp	Se produce cuando el usuario suelta el botón del mouse sobre el control.
MouseWheel	Se produce cuando el usuario mueve la rueda del mouse.

***Tabla 4.** Eventos de los controles con representación gráfica.*

FORMULARIOS

Si hacemos doble clic en **Form1.cs** en la pestaña **Solution Explorer**, veremos el formulario principal de la aplicación en tiempo de diseño (**vista de diseño del formulario**). Aquí podremos agregar otros controles del **Toolbox**, y modificar las propiedades del formulario y de los controles. Las propiedades más importantes son:

ALTERNATIVAS A VISUAL STUDIO .NET

Visual Studio .NET es un excelente entorno de programación, pero también existen alternativas. En el sitio **.net coders** (**www.dotnetcoders.com/web/Articles/ShowArticle.aspx?article=49**) encontraremos un listado con otros entornos que son opciones a la herramienta de Microsoft.

PROPIEDAD	DESCRIPCIÓN
AcceptButton	Indica qué botón del formulario es presionado si el usuario pulsa la tecla Enter.
BackColor	Color de fondo del formulario.
CancelButton	Indica qué botón del formulario es presionado si el usuario pulsa la tecla Escape.
Font	Tipografía predeterminada que empleará el formulario y los controles que contenga.
ForeColor	Color de la tipografía del formulario.
Text	Título del formulario.
Location	Ubicación del formulario en la pantalla.
MaximunSize	Tamaño máximo del formulario (en píxeles de alto y ancho). Si estas propiedades se encuentran en (0, 0), entonces el formulario no tendrá tamaño máximo.
MinimumSize	Tamaño mínimo del formulario (en píxeles de alto y ancho). Si estas propiedades se encuentran en (0, 0), entonces el formulario no tendrá tamaño mínimo.
Size	Tamaño del formulario.
StartPosition	Puede ser: -Manual: la posición es especificada por la propiedad Location. -CenterScreen: se centra la ventana en relación con la pantalla. -WindowsDefaultLocation: posición predeterminada. -WindowsDefaultBounds: posición y tamaño predeterminados. -CenterParent: se centra la ventana en relación con la ventana padre.
WindowState	Puede ser: -Normal: ventana en modo normal. -Minimized: ventana en modo minimizado. -Maximized: ventana en modo maximizado.
ControlBox	Indica si la ventana poseerá los botones de minimizar, maximizar, cerrar y el menú contextual que se desprende de ella cuando hacemos clic sobre su extremo superior izquierdo.
Icon	Icono que representará la ventana.
MaximizeBox	Indica si la ventana poseerá botón de maximizado.
Menu	Indica cuál es el menú de la ventana (luego veremos esta propiedad en detalle).
MinimizeBox	Indica si la ventana poseerá botón de minimizado.
Opacity	Opacidad de la ventana. Mediante esta propiedad podremos volverla semitransparente.
TopMost	Indica si la ventana se encuentra delante de todas las demás, más allá de estar o no seleccionada.

***Tabla 5.** Propiedades del formulario.*

Los métodos más comunes del formulario son:

MÉTODO	DESCRIPCIÓN
Close	Cierra el formulario.
Hide	Oculta el formulario.
Show	Muestra el formulario.

***Tabla 6.** Métodos del formulario.*

Los eventos más comunes del formulario son:

EVENTO	DESCRIPCIÓN
Activated	Es invocado cuando el formulario es activado de algún modo.
Closed	Es invocado inmediatamente después de que el formulario es cerrado.
Closing	Es invocado justo antes de que el formulario sea cerrado.
Deactivated	Es invocado cuando el formulario pierde el foco (deja de ser el formulario activo).
Load	Es invocado antes de que el formulario sea visualizado por primera vez.

***Tabla 7.** Eventos del formulario.*

Cada vez que modificamos el valor predeterminado de alguna propiedad del formulario o de algún control del formulario, **Visual Studio** agrega código al método **InitializeComponent** que mencionamos antes. Podemos verificar esto modificando, por ejemplo, la propiedad **Text** del formulario. Inmediatamente después observaremos, en el método mencionado, el siguiente código:

```
private void InitializeComponent()
{
    // ...
    // Form1
    // ...
    this.Text = "Formulario principal";
}
```

Cuando comencemos a agregar controles sobre el formulario, este método no demorará en hacerse realmente extenso, por lo tanto, es una buena idea colocarlo dentro de un bloque **region**.

CREAR UNA APLICACIÓN SENCILLA

Comenzaremos a modificar nuestra aplicación creada por el asistente de **Visual Studio** agregando dos controles al formulario.

Para esto deberemos acceder a la vista de diseño del formulario y luego hacer visible la barra de herramientas llamada **Toolbox** (debería estar a la derecha o a la izquierda del formulario cuando éste se encuentre en diseño).

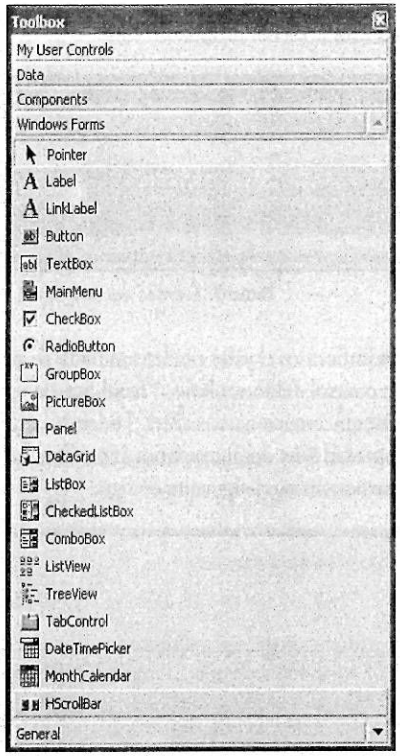


***Figura 7.** La caja de herramientas.*

Ahora, agregaremos a nuestra aplicación dos controles: un **TextBox** y un **Button**.

Para esto seleccionamos el **TextBox** del **Toolbox** y lo insertamos en el formulario. Lo mismo hacemos con el **Button**, ajustando ambos controles hasta lograr el aspecto que deseemos. Podemos cambiar sus nombres si así lo deseamos, o la leyenda predeterminada que muestra el **TextBox** y la que exhibe el **Button**.

Hecho esto, deberíamos estar ante nuestra ventana con un aspecto similar a éste:

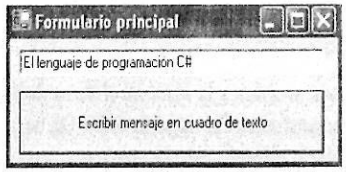


***Figura 8.** Agregando dos controles.*

Ahora, cuando presionamos el botón, querríamos cambiar el contenido del cuadro de texto. Requerimos que el framework **nos avise** cuando se produce la ocurrencia **"Se ha hecho un clic sobre el botón"**. ¿Cómo podríamos hacer esto? ¡Con eventos! Así es, los delegados y los eventos entran nuevamente en acción. Y con **Visual Studio**, la manera de fijarlos es muy sencilla: la página de propiedades posee un botón que conmuta a ver los eventos del control seleccionado (**Figura 9**).

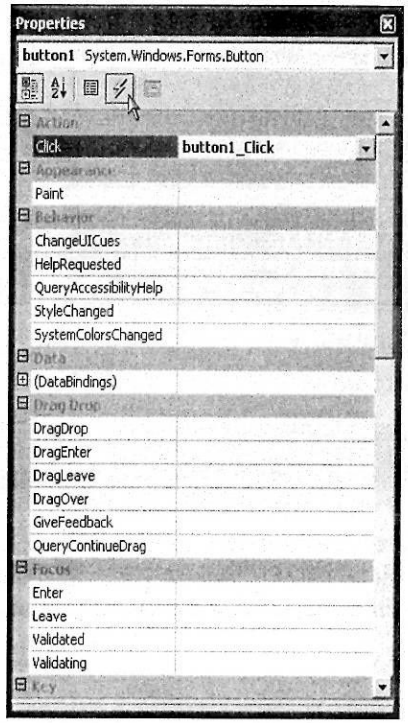


***Figura 9.** Selección de eventos en la página de propiedades (Properties).*

III LA "PROPIEDAD" (NAME)

Cuando modifiquemos la propiedad **(Name)** desde el **Toolbox** del editor de formularios, **Visual Studio** modificará el nombre de nuestro objeto instanciado a partir de la clase del control del formulario. Cualquier referencia a él dentro del código deberá ser modificada manualmente.

Nótese que ahora la página muestra los eventos del control activo (en este caso, el botón). Si hacemos doble clic sobre el evento llamado **Click**, entonces automáticamente generará el código requerido para que un método de nuestro formulario maneje dicha ocurrencia. El código agregado por el diseñador de formularios será:

- El método que trate el evento con identificador conformado por el nombre del control, seguido por el signo *underscore* y del nombre del evento por tratar (como por ejemplo: **button1_Click**).
- Agregado del método al evento por tratar dentro del método **InitializeComponent**.

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Luego de que el entorno haya agregado el código correspondiente, nos situará al comienzo del método que trata el evento para que podamos escribir el código específico que deseemos.

En nuestro caso queremos modificar el valor de la propiedad **Text** del control **TextBox**, por lo tanto escribiremos:

```
textBox1.Text = "El lenguaje de programación C#";
```

Nótese que nuestro **Button** se llama **button1** y nuestro **TextBox** se llama **textbox1**. Éstos son los nombres de las instancias (objetos) que ha fijado el entorno automáticamente. Nosotros podremos hacer los cambios que consideremos adecuados.

Convención de nombramiento para controles

Más adelante nuestros controles serán nombrados haciendo uso de una convención de nombramiento que consiste en anteponer a los nombres de los controles un prefijo de –generalmente– tres letras, correspondiente al tipo de control. Luego, se prosigue con el nombre que mejor describa lo que el control hace, sin separaciones con caracteres especiales, sino especificando en mayúsculas los nombres de las palabras que lo conforman. Por ejemplo:

- Botón de **Ok**: **cmdOk**.
- Botón que imprime una ficha de cliente: **cmdImprimirCliente**.
- **TextBox** que alberga el nombre de un usuario: **txtUsuario**.
- **ListView** que muestra procesos: **lvwProcesos**.

La siguiente es una lista de prefijos en función del tipo de control. No es un deber seguir esta convención, pero muchos programadores lo hacen y facilita en gran medida la legibilidad del código.

TIPO DE CONTROL	PREFIJO
Button	btn
CheckBox	chk
CheckedListBox	chkl
ColorDialog	clrdlg
ComboBox	cbo
ContextMenu	cmnu
Control	ctr
DataGrid	dgr
DataList	dlst
DateTimePicker	dtp
DropDownList	cbo
FontDialog	fntdlg
Form	frm
GroupBox	grp
HelpProvider	hpv
HscrollBar	hsb
Image	img
ImageList	ils
Label	lbl
LinkLabel	llbl
ListBox	lst
ListView	lvw
Menu	mnu
MonthCalendar	mclr
NotifyIcon	nicn
NumericUpDown	nudn
OpenFileDialog	ofdlg
PageSetupDialog	psdlg
Panel	pnl
PictureBox	pic
PrintDialog	prtdlg
PrintDocument	prtdoc
ProgressBar	prg
RadioButton	rdb
RadioButtonList	rdbl
RichTextBox	rtf

TIPO DE CONTROL	PREFIJO
SaveFileDialog	sdlg
Splitter	spt
StatusBar	stb
TabControl	tab
TextBox	txt
Timer	tmr
ToolBar	tlb
ToolTip	tlt
TrackBar	tbr
TreeView	tvw
VscrollBar	vsb

***Tabla 8.** Prefijos según el tipo de control.*

MANIPULACIÓN DE LOS CONTROLES BÁSICOS

Veremos ahora cómo realizar una aplicación que posea controles del tipo: **TextBox**, **Button** y **ListBox**.

Caso de estudio 1

Controles empleados: TextBox, ListBox, Button.

Crearemos una aplicación con un cuadro de texto, un cuadro de lista y tres botones. La idea será que, presionando un botón, se agregue el texto del cuadro a la lista, con el segundo botón se elimine la información seleccionada en la lista y con el tercer botón se limpie la lista por completo. La aplicación tendrá el siguiente aspecto:

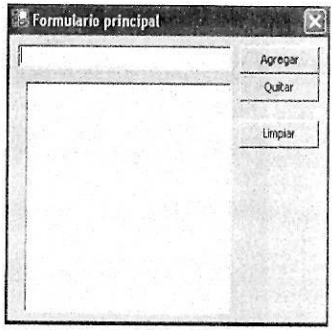


***Figura 10.** Haciendo uso del ListBox.*

TextBox

Sus propiedades más importantes son:

PROPIEDAD	DESCRIPCIÓN
MaxLength	Máxima cantidad de caracteres del cuadro.
Multiline	Indica si el control es multilínea.
ReadOnly	Si es verdadero, impide que el usuario pueda modificar el contenido del texto dentro del cuadro.
Text	Indica el contenido de texto del cuadro.
TextAlign	Indica la alineación del texto dentro del cuadro.
PasswordChar	Indica el signo que simbolizará un carácter en una clave.

***Tabla 9.** Propiedades del TextBox.*

Sus métodos más importantes son:

MÉTODO	DESCRIPCIÓN
Clear	Limpia el control.

***Tabla 10.** Método del TextBox.*

Sus eventos más importantes son:

EVENTO	DESCRIPCIÓN
TextChanged	Es invocado cuando cambia el texto dentro del cuadro.

***Tabla 11.** Evento del TextBox.*

Button

El botón lo utilizaremos para disparar las acciones. Simplemente nos **colgamos** del evento **Click** como lo hemos hecho anteriormente.
Sus propiedades son:

PROPIEDAD	DESCRIPCIÓN
Image	Indica la imagen que contendrá el botón (si es que poseerá alguna).
ImageIndex	Indica el índice de la imagen si es que el botón posee un ImageList relacionado.
Text	Indica el contenido de texto del botón.
TextAlign	Indica la alineación del texto dentro del botón.

***Tabla 12.** Propiedades del Button.*

Sus eventos más importantes son:

EVENTO	DESCRIPCIÓN
Click	Es invocado cuando se presiona el botón izquierdo del mouse sobre el botón.

***Tabla 13.** Evento del Button.*

ListBox

Existen controles que no manejan un solo dato, sino **un conjunto de datos**. Como vimos, la librería **BCL** provee un excelente soporte a las colecciones. Por lo tanto, los controles que manipulan conjuntos de datos usan varias de las interfaces de colecciones que ofrece la librería, para trabajar con el conjunto de datos que poseen. Esto es bueno, ya que muchas de las operaciones nos serán conocidas.

Tal vez la propiedad más importante que posee el control **ListBox** es **Items**. **Items** es una propiedad del tipo **ListBox.ObjectCollection** que implementa las interfaces **IList**, **ICollection** e **IEnumerable**; esto significa que dispondremos de los métodos implementados de estas interfaces para manipular el contenido de la lista.

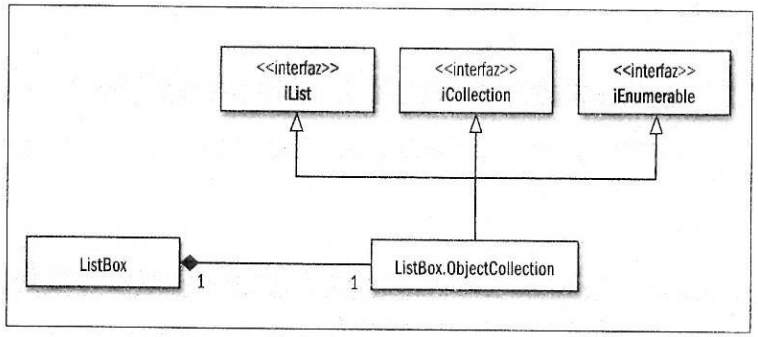


***Figura 11.** El diagrama de clases simplificado del control ListBox.*

Entonces, para agregar un elemento a la lista deberemos utilizar el método **Add** de la propiedad **Items** de la lista. Para eliminar el contenido de la lista deberemos utilizar el método **Clear** de la misma propiedad, y, finalmente, para eliminar un ítem deberemos utilizar el método **Remove**.

El método Add

El método **Add** recibe un objeto de tipo **Object** como parámetro. Luego, como hemos visto, todo **Object** debe implementar un método **ToString()** que devuelva una

OTROS CONTROLES

En la Red encontraremos muchos sitios que ofrecen librerías de controles para bajar e integrar muy fácilmente a nuestras aplicaciones. **ComponentOne** es uno de los más conocidos en este aspecto (**www.componentone.com**).

representación del contenido del objeto en formato texto. Dicho texto será el que exhibirá el **ListBox**.

Además, el hecho de que toda clase sea descendiente de **Object** resulta en que el **ListBox** puede almacenar en su interior cualquier tipo de objeto definido por nosotros.

El código que especificaremos para agregar texto a nuestra lista será:

```
lstUsuarios.Items.Add(txtUsuario.Text);
```

El método Clear

Utilizar el método **Clear** es muy sencillo, no posee parámetros.
Simplemente escribiremos:

```
lstUsuarios.Items.Clear();
```

El método Remove

El método **Remove** recibe un objeto de tipo **Object** como parámetro. Este objeto será el que la lista misma nos informe que está seleccionado.
Escribiremos, entonces:

```
lstUsuarios.Items.Remove(lstUsuarios.SelectedItem);
```

Ahora, agregaremos una verificación extra. No deseamos que el usuario pueda agregar a nuestra lista dos nombres iguales. Si esto ocurre, se deberá informar esta situación mediante un cuadro de mensaje. Tampoco deseamos introducir valores nulos.

Por lo tanto, el código que invoca el evento **Click** de nuestro primer botón será:

```
if (txtUsuario.Text == "")
    return;
if (lstUsuarios.Items.Contains(txtUsuario.Text) == false)
    lstUsuarios.Items.Add(txtUsuario.Text);
else
    MessageBox.Show("El valor ya se encuentra en la lista");
```

Otras propiedades del control **ListBox**:

PROPIEDAD	DESCRIPCIÓN
Items	Colección de elementos de la lista.
SelectedIndex	Indica cuál es el índice del ítem seleccionado.
SelectedIndices	Indica cuáles son los índices de los ítems seleccionados.
SelectedItem	Indica cuál es el ítem seleccionado.
SelectedItems	Indica cuáles son los elementos seleccionados.
SelectedValue	Indica cuál es el valor del ítem seleccionado.
Sorted	Indica si la lista se encuentra ordenada.

***Tabla 14.** Más propiedades del ListBox.*

Otros métodos importantes:

- **FindString**: busca el primer ítem de la lista que comienza con el texto especificado.

Otros eventos importantes:

- **SelectedIndexChanged**: invocado cuando la propiedad **SelectedIndex** se modifica.
- **SelectedValueChanged**: invocado cuando la propiedad **SelectedValue** se modifica.

Sin embargo, mejor sería que si el texto del cuadro **txtUsuario** fuera nulo, el usuario ni siquiera pudiera presionar el botón **Agregar**. ¿Cómo podríamos hacer esto? Existe una propiedad de casi todos los controles llamada **Enabled**, que permite habilitar o deshabilitar un control. Luego, los cuadros de texto poseen un evento llamado **TextChanged**, que es invocado cuando existe una modificación sobre el contenido del cuadro.

Utilizaremos la propiedad y el evento citados para cumplir nuestro cometido.

1. El botón **cmdAgregar** ahora comenzará con valor **false** en la propiedad **Enabled**.
2. Nos **colgaremos** del evento **TextChanged** del cuadro de texto **txtUsuario** y escribiremos la siguiente línea de código:

```
cmdAgregar.Enabled = !(txtUsuario.Text == "");
```

Con esto queremos afirmar que la propiedad **Enabled** de **cmdAgregar** será la negación de la expresión booleana **txtUsuario.Text == ""**. En consecuencia, si el cuadro de texto **sí** está vacío, el botón **no** se habilita, en tanto que si el cuadro de texto **no** se encuentra vacío, el botón **sí** se habilita.

De un modo similar procederemos con los botones **cmdQuitar** y **cmdLimpiar**. No queremos ofrecerle al usuario la posibilidad de hacer algo que no tiene sentido que ocurra. ¿Por qué habilitar el botón **Quitar** cuando no existe ningún elemento seleccionado en la lista?, o ¿por qué ofrecer la posibilidad de limpiar la lista cuando se encuentra vacía?

Implementar estas verificaciones es muy sencillo:

1. Cada vez que se agrega un elemento, se habilita el botón **Limpiar**.
2. Cada vez que se quita un elemento: a) se verifica que la lista no haya quedado vacía; b) se verifica que exista alguna selección válida.
3. Cada vez que se limpia la lista: a) se deshabilita el botón **Limpiar**; b) se deshabilita el botón **Quitar**.
4. Cuando se modifica en la lista el elemento seleccionado activo (evento **SelectValueChanged**), verificamos si debemos activar el botón **Quitar**.

El código completo de esta aplicación lo podemos encontrar en el proyecto **GUIApp1**, que se puede descargar desde el sitio **onweb.tectimes.com**.

Caso de estudio 2

Controles empleados: **RadioButton**, **GroupBox** y **CheckBox**.

A continuación prestemos atención a cómo utilizar los botones de opción, los cuadros de verificación y los cuadros de grupo.
La aplicación es muy simple. Solamente colocaremos código en los eventos más comunes de los controles y utilizaremos sus propiedades básicas.

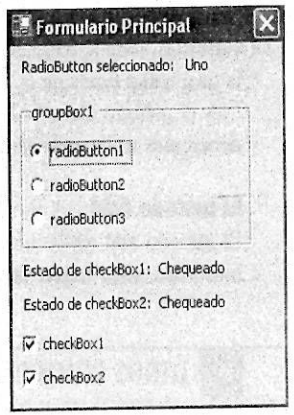


***Figura 12.** El manejo de botones de opción y cuadros de verificación es muy sencillo.*

Las propiedades que marcan o desmarcan los botones de opción, así como los cuadros de verificación, es **Checked**. Esta opción es booleana, por lo que acepta dos valores: verdadero (**true**) o falso (**false**).

El cuadro de verificación, sin embargo, podría quedar en un tercer estado, que es indeterminado. Para poder fijar el control en este estado particular utilizaremos la propiedad **CheckState**. Ésta acepta los valores **Checked**, **Unchecked** e **Indeterminate**.

El cuadro de grupo se suele utilizar para agrupar controles. Si bien posee sus propiedades y algunos eventos, en general éstos no varían en tiempo de ejecución.

El evento que se invoca cuando el control cuadro de opción o cuadro de verificación es modificado se llama **CheckedChanged**. En resumen, el código que escribiremos será, principalmente:

```
private void rdb1_CheckedChanged(object sender, System.EventArgs e)
{
     lblRBSeleccionado.Text = "Uno";
}
private void rdb2_CheckedChanged(object sender, System.EventArgs e)
{
     lblRBSeleccionado.Text = "Dos";
}
private void rdb3_CheckedChanged(object sender, System.EventArgs e)
{
     lblRBSeleccionado.Text = "Tres";
}
private void chk1_CheckedChanged(object sender, System.EventArgs e)
{
     if (chk1.Checked)
          lblEstadoCB1.Text = "Chequeado";
     else
          lblEstadoCB1.Text = "No chequeado";
}
private void chk2_CheckedChanged(object sender, System.EventArgs e)
{
     if (chk2.Checked)
          lblEstadoCB2.Text = "Chequeado";
     else
          lblEstadoCB2.Text = "No chequeado";
}
```

Podemos encontrar el código completo de esta aplicación en el proyecto **GUIApp2**, que se puede descargar desde el sitio oficial del libro (**onweb.tectimes.com**).

Caso de estudio 3

Controles empleados: ListView, ImageList.

El control **ListView** es muy empleado gracias a su flexibilidad. De algún modo reemplaza al clásico **ListBox**, que ahora sólo se utiliza en casos muy sencillos. Es que el **ListView** nos permite crear listas con múltiples columnas, iconos representativos, diversas vistas de la información, etc. Sus propiedades más importantes son:

PROPIEDAD	DESCRIPCIÓN
Columns	Colección de columnas.
FullRowSelect	Indica si la selección es por toda la fila.
GridLines	Indica si se deben mostrar las líneas de guía.
Items	Colección de elementos de la lista.
LabelEdit	Indica si se permite la edición de elementos sobre la lista.
SelectedIndices	Indica cuáles son los índices de los ítems seleccionados.
SelectedItems	Indica cuáles son los elementos seleccionados.

***Tabla 15.** Propiedades del control ListView.*

Sus métodos más importantes:

MÉTODO	DESCRIPCIÓN
Clear	Elimina todos los elementos y columnas del control (no confundir con el Clear de la propiedad Items).
Sort	Ordena los elementos de la lista.

***Tabla 16.** Métodos del control ListView.*

Sus eventos más importantes:

EVENTO	DESCRIPCIÓN
ColumnClick	Es invocado cuando el usuario realiza un clic sobre alguna columna.
ItemCheck	Es invocado cuando cambia el estado de verificación de algún elemento.
SelectedIndexChanged	Es invocado cuando la propiedad SelectedIndex se modifica.

***Tabla 17.** Eventos del control ListView.*

III PROPIEDADES COMUNES

Gracias al buen diseño de la librería BCL, muchas de las propiedades que encontraremos en controles que nunca antes hemos utilizado nos serán familiares. Si al control se le puede fijar un texto, seguramente se realizará por medio de la propiedad **Text**. Esto no ocurría en versiones anteriores de Visual Basic ni en la librería VCL de Borland.

Observemos una aplicación de ejemplo sencilla en la cual veremos un conjunto de datos, que en este caso serán nombres de archivos de un directorio, y que nos permita jugar con algunas de las propiedades de la lista.

La manipulación de los datos internos del **ListView** es similar al **ListBox**; aquí también existe una propiedad que implementa las interfaces **IList**, **ICollection** e **IEnumerable**. Excepto que ahora la propiedad **Items** será del tipo **ListView.ListViewItemCollection**.

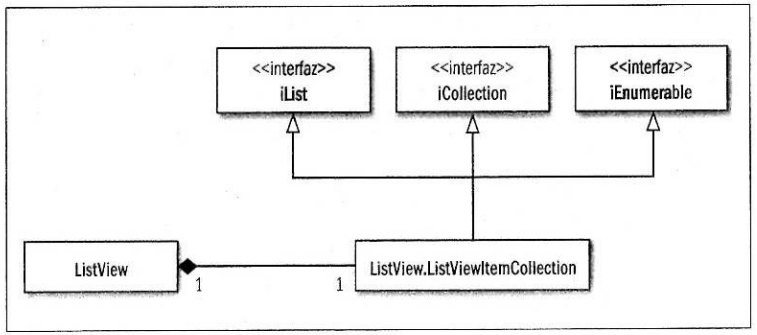


Figura 13. *Diagrama de clases simplificado del control ListView.*

Los métodos que encontraremos en esta implementación de colección serán, por supuesto, los declarados en las interfaces, pero cabe hacer una mención: el método **Add** no recibe un objeto del tipo **Object**, sino que espera un objeto del tipo **ListViewItem**; y es que ahora no existe sólo una columna, sino varias (o al menos existe la posibilidad de que esto suceda). También podremos hacer un uso trivial de la lista aprovechando una versión de **Add** que recibe un string, escribiendo:

```
lvwArchivos.Items.Add("Este es el item 1");
lvwArchivos.Items.Add("Este es el item 2");
lvwArchivos.Items.Add("Este es el item 3");
lvwArchivos.Items.Add("Este es el item 4");
```

III NUESTROS PROPIOS CONTROLES

Crear un control nuevo es muy sencillo. El asistente de aplicaciones generará el marco de la librería que requeriremos, y sólo deberemos encargarnos de escribir código específico del tipo de control que deseemos crear.

Además, antes de utilizar el control deberemos especificar las columnas que poseerá. Esto se puede realizar en tiempo de diseño (propiedad **Columns**):

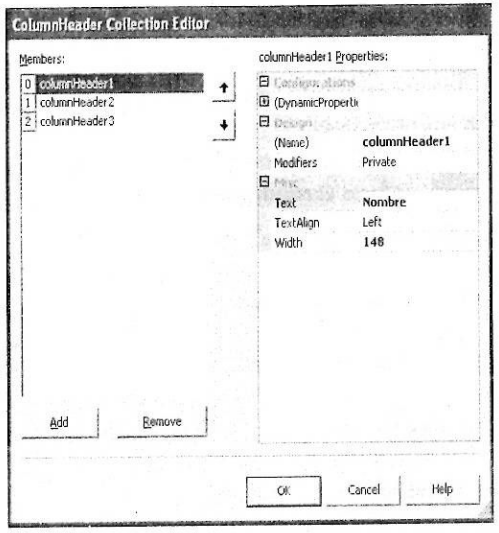


Figura 14. *Agregando columnas al ListView.*

Y también podríamos hacerlo mediante código:

```
lvwArchivos.Columns.Clear();
ColumnHeader ch;
ch = new ColumnHeader();
ch.Text = "Columna 1";
ch.Width = 100;
lvwArchivos.Columns.Add(ch);
ch = new ColumnHeader();
ch.Text = "Columna 2";
ch.Width = 100;
lvwArchivos.Columns.Add(ch);
ch = new ColumnHeader();
ch.Text = "Columna 3";
ch.Width = 100;
lvwArchivos.Columns.Add(ch);
```

La propiedad **Columns** es del tipo **ListView.ColumnHeaderCollection**, que también implementa **IList**, **ICollection** e **IEnumerable**. Luego, en caso de que deseemos agregar

columnas, el método **Add** acepta objetos del tipo **ColumnHeader**, por lo que debemos crear dicho objeto con anterioridad.

Notemos que en el listado anterior no reutilizamos el objeto **ch** para todas las columnas, sino que creamos objetos nuevos. El asunto es que aquí el control almacena referencias a las columnas, y si utilizamos los mismos objetos, el control no se comportará como esperamos que lo haga.

Respecto a las filas, el modo de manejarnos será similar:

```
ListViewItem lvi;
lvi = new ListViewItem("primera fila, primera columna");
lvi.SubItems.Add("primera fila, segunda columna");
lvi.SubItems.Add("primera fila, tercera columna");
lvwProcesos.Items.Add(lvi);
lvi = new ListViewItem("segunda fila, primera columna");
lvi.SubItems.Add("segunda fila, segunda columna");
lvi.SubItems.Add("segunda fila, tercera columna");
lvwProcesos.Items.Add(lvi);
// ...
```

Aquí crearemos un objeto del tipo **ListViewItem**, al que agregaremos subelementos utilizando para ello el método **SubItems.Add** y luego lo relacionaremos con la lista por medio de su método **Add**.

Con la única finalidad de hacer un poco más atractiva la aplicación, ingresaremos los datos al control **ListView** a partir de la información que obtengamos del listado de archivos de un directorio.

Haciendo uso de la clase **System.IO.DirectoryInfo** podremos obtener la información que deseamos sobre el directorio.

Luego, relacionaremos los datos de las filas con un icono representativo. Para esto deberemos agregar a nuestro formulario un control del tipo **ImageList**. Este control es un contenedor de imágenes que se puede relacionar con otros controles.

Para relacionar nuestro **ListView** con nuestro flamante **ImageList**, deberemos acceder a la propiedad **SmallImageList**, que posee el **ListView** desde la vista de diseño de formularios. Esta propiedad es especificada seleccionando un elemento de la lista que nos ofrece. Si la desplegamos, podremos observar que nos ofrece todos los

nombres de los **ImageList** que posee nuestro formulario. Una vez que seleccionamos el único que hemos agregado, la relación ha sido consumada.

Las imágenes dentro del **ImageList** poseen un índice, deberemos referirnos a alguna de ellas por medio de este número. En nuestro ejemplo, hemos agregado un pequeño bitmap de 16x16 al **ImageList** que referenciaremos por su índice: el número cero.

```
ListViewItem lvi;
lvi = new ListViewItem("primer fila, primer columna");
lvi.SubItems.Add("primer fila, segunda columna");
lvi.SubItems.Add("primer fila, tercer columna");
lvi.ImageIndex = 0;
lvwProcesos.Items.Add(lvi);
```

Como se puede apreciar en el listado anterior, antes de agregar el objeto del tipo **ListViewItem** al **ListView**, especificamos que su **ImageIndex** sea 0.

Finalmente, cabe señalar que este tipo de listas posee distintos modos de visualización (que ya conocemos, por usar diariamente el explorador de archivos de Windows). Existe el modo **Details**, donde se observan todas las columnas y la información en modo de filas; el modo **SmallIcon**, donde se aprecian los elementos sólo por el nombre ingresado en la primera columna y con un icono pequeño que le hemos relacionado; el modo **LargeIcon**, que es similar al **SmallIcon**, aunque utiliza un icono más grande que también deberíamos relacionarle; y finalmente el modo **List**, que muestra la lista como **Details** pero ocultando todas las columnas menos la primera.

En el ejemplo usamos un control de tipo **ComboBox** para elegir la vista del **ListView**.

También colocamos un **CheckBox** para modificar la propiedad **GridLines** del **ListView**. Con **GridLines** activado, la lista muestra unas líneas de guías horizontales y verticales (sólo es visible en modo **Details**).

III CONTROLES CONOCIDOS

Los controles que ofrece la librería BCL son los que podremos encontrar en cualquier otra aplicación convencional de Windows, y son lo suficientemente vastos como para adaptarse a muchísimas necesidades.

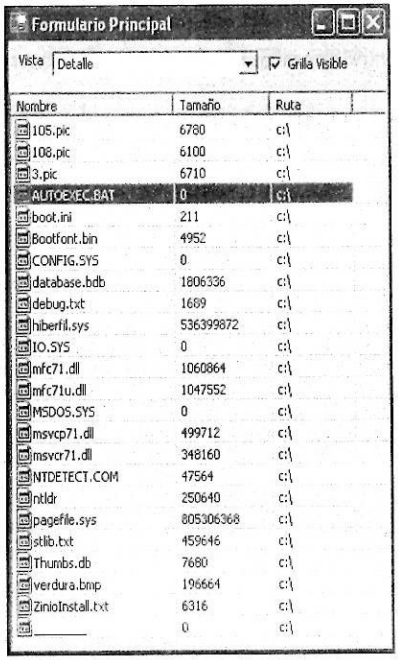


***Figura 15.** El control TreeView mostrando los datos que le ingresamos.*

Podemos encontrar el código completo de esta aplicación en el proyecto **GUIApp3**, que se puede descargar desde el sitio oficial del libro.

Caso de estudio 4

Controles empleados: TreeView, ImageList.

El control **TreeView** es muy práctico y poderoso. Es la pareja ideal para el **ListView**, y muchísimas veces se utilizan en conjunto. Este control nos permite mostrar la información de un modo jerárquico, es decir que los elementos del control pueden depender de otros elementos (como archivos dentro de carpetas).

Al igual que el **ListView**, el **TreeView** permite asociar imágenes de un **ImageList** a sus elementos. El modo en que se relacionan ambos controles es exactamente el mismo que el que ya hemos analizado.

También, además de muchos otros controles que manejan un conjunto de datos, posee una propiedad que implementa las interfaces **iList**, **iCollection** e **iEnumerable**.

Esta propiedad es **Nodes** y pertenece a la clase **TreeNodeCollection**.

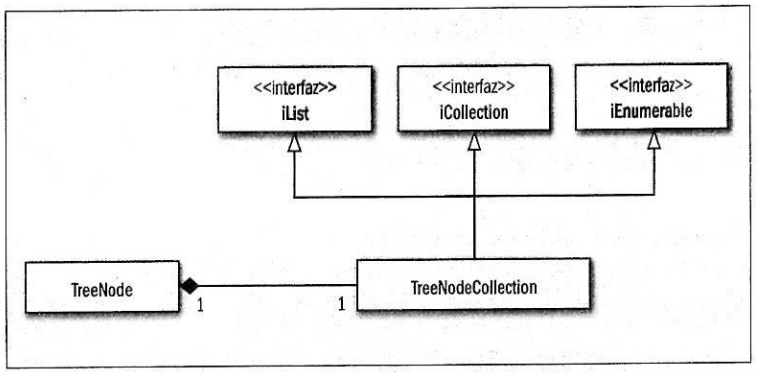


***Figura 16.** Diagrama de clases simplificado de TreeView.*

Por medio de **Nodes** manipularemos la información de los datos que muestre el **TreeView**. Si deseamos agregar un elemento, deberemos escribir:

```
TreeNode tn1 = new TreeNode("Uno", 0, 0);
tvwElementos.Nodes.Add(tn1);
```

El primer parámetro que recibe el constructor de **TreeNode** es el texto que representará el nodo; el segundo es un índice de una imagen del **ImageList** relacionado que se utilizará como icono para el nodo en cuestión, y el tercer parámetro también es un índice de imagen, pero será empleado sólo cuando el usuario esté seleccionando el nodo (en nuestro caso utilizamos la misma imagen, pero podríamos cambiarla para dar un efecto estético distinto).

Luego, con el objeto de tipo **TreeNode** creado, simplemente haremos uso del método **Add** de la propiedad **Nodes** de la clase **TreeView**.

MEJORANDO LA INTERFAZ DE USUARIO

Antes de crear la interfaz de usuario de nuestra aplicación, será conveniente conocer un poco la ciencia que subyace tras su diseño. Existen libros que tratan este tema, y varios artículos que podremos encontrar en la Red, como en **www.medicalcomputingtoday.com/0agui.html**.

Si observamos cautelosamente, lograremos advertir que la clase **TreeNode** también posee una propiedad **Nodes**.
Esto quiere decir que cada nodo puede contener otros nodos, y precisamente esto es lo interesante del control **TreeView**: la organización jerárquica de los elementos que lo componen.

Entonces, si deseamos crear un subnodo del nodo **tn1**, podremos escribir:

```
TreeNode tn2 = new TreeNode("Dos", 1, 1);
tn1.Nodes.Add(tn2);
```

Téngase en cuenta la pequeña y sutil diferencia que posee este listado en comparación con el anterior: en este momento no utilizamos el objeto **tvwElementos** (que es el nombre de nuestro objeto de tipo **TreeView**), sino **tn1**, que se trata de un objeto de tipo **TreeNode**.

Esta operación también podría repetirse con **tn2**, es decir que **cada nodo** puede poseer uno o más nodos hijos sin que el control especifique límite alguno.

Para limpiar los elementos de la colección no hay novedades; deberemos escribir:

```
tvwElementos.Nodes.Clear();
```

Luego existen algunas otras propiedades de interés de la clase **TreeView**, que son:

PROPIEDAD	DESCRIPCIÓN
CheckBoxes	Indica si se deben mostrar cuadros de verificación junto a los nodos.
ImageIndex	Índice de imagen predeterminado que se utilizará en los nodos.
Indent	Distancia desde el margen izquierdo hasta los nodos.
LabelEdit	Indica si se permite editar el texto de los nodos.
SelectedImageIndex	Índice de imagen de nodo seleccionado por predefinición que se utilizará en los nodos.
ShowLines	Indica si se deben mostrar las líneas de guías.
ShowPlusMinus	Indica si se deben mostrar los cuadros junto a los nodos que indican si éste se encuentra expandido o contraído.
ShowRootLines	Indica si se debe mostrar la línea raíz del árbol.
TopNode	Devuelve el primer nodo visible del control.
VisibleCount	Devuelve la cantidad de nodos visibles que muestra el control.

***Tabla 18.** Propiedades de la clase TreeView.*

Sus métodos más importantes son:

MÉTODO	DESCRIPCIÓN
CollapseAll	Contrae todos los nodos de la lista.
ExpandAll	Expande todos los nodos de la lista (mostrando todos los subnodos).
GetNodeAt	Permite obtener un nodo del TreeView a partir de una especificación de coordenadas en pixeles.
GetNodeCount	Devuelve la cantidad de nodos que posee el TreeView.

***Tabla 19.** Métodos más importantes del control TreeView.*

Y sus eventos más importantes:

EVENTO	DESCRIPCIÓN
AfterCheck	Invocado después de marcar un cuadro de verificación relacionado con un nodo.
AfterCollapse	Invocado después de contraer un nodo.
AfterExpand	Invocado después de expandir un nodo.
AfterSelect	Invocado después de seleccionar un nodo.
BeforeCheck	Invocado antes de marcar un cuadro de verificación relacionado con un nodo.
BeforeCollapse	Invocado antes de contraer un nodo.
BeforeExpand	Invocado antes de expandir un nodo.
BeforeSelect	Invocado antes de seleccionar un nodo.

***Tabla 20.** Eventos de la clase TreeView.*

La aplicación de ejemplo que desarrollaremos a continuación tendrá la función de realizar un esquema de elementos arbitrario y modificará algunas de las propiedades pertenecientes al control.

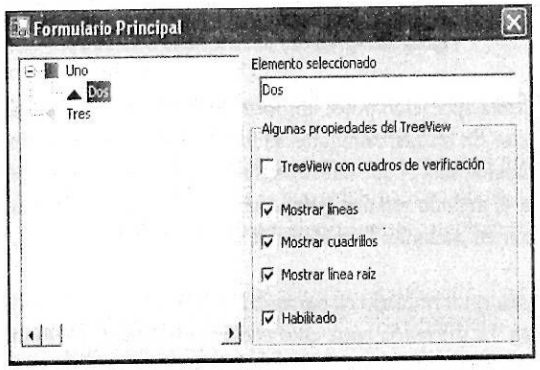


***Figura 17.** La aplicación del ejemplo en acción.*

Podemos encontrar el código completo de esta aplicación en el proyecto **GUIApp4**, el cual se puede descargar desde el sitio oficial del libro: **onweb.tectimes.com**.

Caso de estudio 5

Controles empleados: múltiples formularios, **Menu**, **TrackBar**, **ProgressBar**.

Hasta ahora, nuestras aplicaciones tenían sólo un formulario. Pero usualmente las aplicaciones poseen más de uno. Para agregar un nuevo formulario a nuestra aplicación, sólo deberemos seleccionar nuestro proyecto desde el **Solution Explorer**, presionar el botón derecho del mouse e ingresar en la opción **Add/Add Windows Form...**.

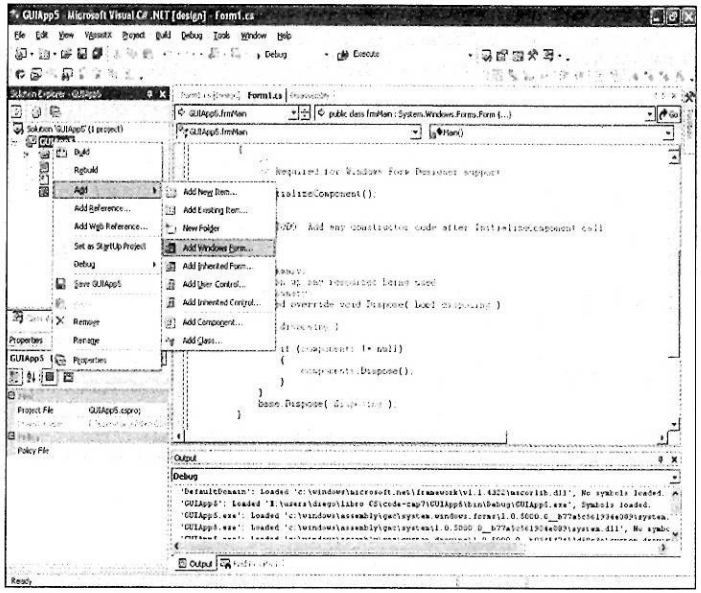


Figura 18. *Agregando un formulario a nuestro proyecto.*

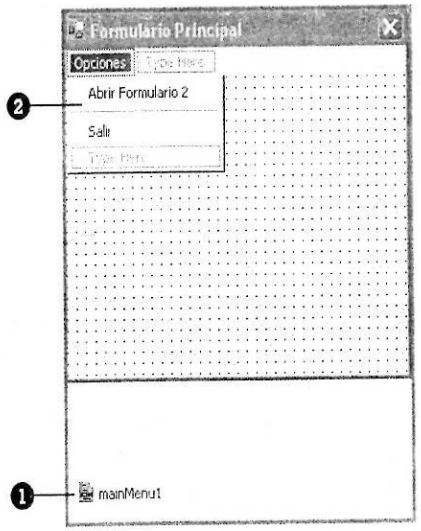
Ahora nuestra aplicación posee dos formularios. ¿Cuál es el principal? ¿El primero? ¿El segundo? En realidad, ninguno. El formulario que se verá cuando el usuario inicie la aplicación será el que mostremos nosotros, y esto actualmente lo estamos haciendo en el método estático **Main**, que se encuentra en el formulario **Form1** pero podría estar en cualquier formulario de nuestra aplicación.

Para mostrar un formulario en particular, sólo deberemos invocar el método **Show** o **ShowDialog**. La diferencia entre ellos consiste en el modo de mostrar el formulario. **Show** abre el formulario de modo **no modal**, es decir que el usuario podrá seguir operando con otros formularios de nuestra aplicación sin tener que cerrar el formulario que le acabamos de abrir. **ShowDialog**, en cambio, muestra el formulario de un modo **modal**; esto significa que el usuario no podrá seguir interactuando con otros formularios abiertos de nuestra aplicación hasta que se cierre el recién abierto.

En la aplicación de ejemplo, el formulario 2 lo abrimos a partir de un menú que colocamos en el formulario 1. Para esto bastará con seleccionar el control de menú desde el **Toolbox** y luego presionar el botón izquierdo sobre nuestro formulario.

El control **Menu** es representado por un pequeño icono debajo de nuestro formulario (donde generalmente se colocan los controles sin representación gráfica). Y también es visible como menú del formulario. Allí mismo podremos comenzar a editarlo, agregando opciones y eventos a las opciones especificadas.

Control Menu — GUÍA VISUAL 2



❶ Icono que representa al menú.
❷ Menú editable en tiempo de diseño.

Entonces, escribiremos código para el evento **Click** de la opción de menú que especifiquemos para abrir el segundo formulario; este código será:

```
Form2 frm2 = new Form2();
frm2.Show ();
```

Cuando el usuario haga clic sobre la opción de menú, se instanciará el formulario y se mostrará en pantalla.
Para evitar que se cree un objeto nuevo cada vez que el usuario abra el formulario, podríamos agregar a **frm2** como una variable de **Form1** y con un inicializador (haciendo un **new Form2()** en la misma línea en que lo declaramos), y luego simplemente podríamos hacer:

```
frm2.Show();
```

Pero si el usuario no abre nunca el formulario 2, de todos modos lo habremos instanciado al inicio de la aplicación. Tal vez ésta no sea una buena idea; dependerá de la importancia del formulario 2 para la aplicación: si siempre será abierto o no.

Una tercera opción podría ser inicializar la variable **frm2** a **null** y luego, cuando se desee abrir el formulario, escribir:

```
if (frm2 == null)
    frm2 = new Form2();
frm2.Show();
```

Cuando la aplicación se inicie, no se solicitará memoria para **frm2**. Esto sólo se hará la primera vez que el usuario desee abrir dicho formulario. Lo malo de esta solución será que siempre, antes de utilizar alguna propiedad o método de **frm2**, deberemos verificar que no sea **null**, o generaríamos una excepción en tiempo de ejecución.

En nuestro ejemplo, el formulario 2 es muy sencillo. Sólo posee un botón que lo oculta:

```
Hide();
```

También podríamos haberlo cerrado:

```
Close();
```

Pero **frm2** nunca más podría ser abierto; por eso deberemos hacer otro **new Form2()**.

Por último, agregamos dos controles más al formulario 1 para hacer más interesante la cuestión. Éstos son el **TrackBar** y el **ProgressBar**, ambos muy sencillos de usar.

En nuestro ejemplo, nos **colgamos** del evento **Scroll** del **TrackBar** que se invoca cada vez que alguien quiere modificar el valor de la barra (por código o por interacción con el mouse), y dentro de este evento escribimos:

```
prgMedidor.Value = tbrBarra.Value;
```

El valor de un **TrackBar** se lee y se escribe del mismo modo que el control **ProgressBar**: a través de su propiedad **Value**.
Existen otras propiedades generalmente modificadas cuando se utiliza este tipo de controles, que son:

PROPIEDAD	DESCRIPCIÓN
Minimum	Valor mínimo que puede tomar el TrackBar o ProgressBar.
Maximum	Valor máximo que puede tomar el TrackBar o ProgressBar.

***Tabla 21.** Otras propiedades.*

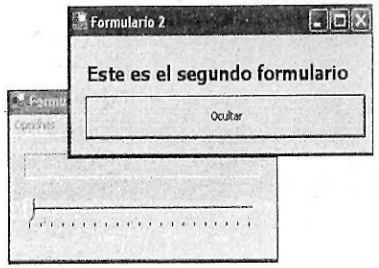


***Figura 19.** Nuestro pequeño ejemplo en acción.*

Podemos encontrar el código completo de esta aplicación en el proyecto **GUIApp5**, que se puede descargar desde el sitio oficial del libro: **onweb.tectimes.com**.

RESUMEN

En este capítulo hemos dado el puntapié inicial en la construcción de aplicaciones con controles. .NET hace esto muy sencillo gracias al excelente diseño que posee la librería BCL. En los próximos capítulos, todos los ejemplos que estudiemos estarán basados en aplicaciones de este tipo, y cuando veamos alguna propiedad o método nuevo relacionado con los controles, lo indicaremos convenientemente.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 Nombrar tres controles que manipulen colecciones y tres que no lo hagan. Nombrar tres controles que no posean representación gráfica en tiempo de ejecución.

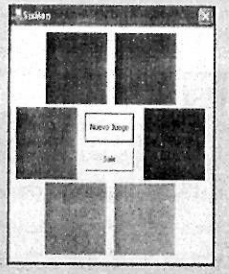
2 ¿Pueden ciertos controles contener a otros?

3 ¿Puede un TextBox poseer en su interior texto de diversas tipografías?

4 ¿Son los valores de los controles de tipo CheckBox dentro de un GroupBox mutuamente excluyentes? ¿Y los controles de tipo RadioButton?

EJERCICIOS PRÁCTICOS

✔ Construir un pequeño juego basado en un formulario con seis botones que posean colores diferentes. El formulario también debe poseer dos botones más: uno para comenzar el juego y otro para cerrar la aplicación.
El juego consiste en generar una secuencia aleatoria en la cual los botones se van encendiendo y apagando (cambiando de color de fondo a uno más claro para generar este efecto), que el jugador debe imitar. Dicha secuencia va extendiéndose progresivamente en cada turno a medida que el jugador acierte.
Entonces, en el primer turno el juego enciende un botón y espera a que el jugador lo imite. Si lo hace bien, vuelve a encender el mismo panel y agrega otro más (avanza sobre la secuencia generada); nuevamente el jugador deberá imitar la secuencia propuesta. El juego termina cuando el jugador consigue imitar el tamaño máximo de la secuencia (este valor debe estar especificado en una constante) o cuando se equivoca. En ambos casos se debe generar un mensaje de texto que informe de la situación.



Capítulo 8

Acceso a datos externos

Un programa que no interactúe con elementos externos no resulta de mucho interés. En este capítulo veremos cómo acceder a archivos de texto y binarios, como también a bases de datos relacionales por medio de ADO.NET.

ARCHIVOS

Todos sabemos que un archivo es una colección de datos situada en algún dispositivo de almacenamiento, y que posee un nombre por medio del cual podemos referenciarlo para interactuar con él.

La librería **BCL** ofrece muchas clases para facilitarnos su manipulación. De éstas, tal vez la más popular sea la clase **File**.

File

La clase **File** se compone sólo por métodos estáticos que nos permiten crear, copiar, eliminar, mover y abrir archivos. Veamos algunos de sus métodos más importantes:

MÉTODO	DESCRIPCIÓN
Copy	Copia un archivo.
Create	Crea un archivo.
Delete	Elimina el archivo especificado como parámetro (por nombre).
Exists	Informa si el archivo pasado como parámetro (por nombre) existe o no.
Move	Mueve un archivo.
GetAttributes	Devuelve los atributos de un archivo.
Open	Abre un archivo.
OpenText	Abre un archivo de texto.
SetAttributes	Fija los atributos de un archivo.

***Tabla 1.** Métodos de la clase File.*

Lo que no posee la clase **File** son métodos para leer o escribir el contenido del archivo. Para esto deberemos utilizar los objetos que nos devuelven los métodos **Create** y **Open**.

Estos métodos nos devolverán diversos tipos de *streams* (flujos) con los cuales sí podremos manipular el contenido de los archivos.

Tipos de streams

Existen diversos tipos de streams:

- **Streams de memoria** (clase **MemoryStream**): la fuente de datos se encuentra situada en la memoria de la computadora.
- **Streams de archivos** (clase **FileStream**): la fuente de datos son archivos.
- **Streams de redes** (clase **NetworkStream**): la fuente de datos es el extremo de una comunicación que se efectúa vía red.

Todos estos tipos de streams descienden de la misma clase **Stream**, como se puede apreciar en el siguiente diagrama de clases:

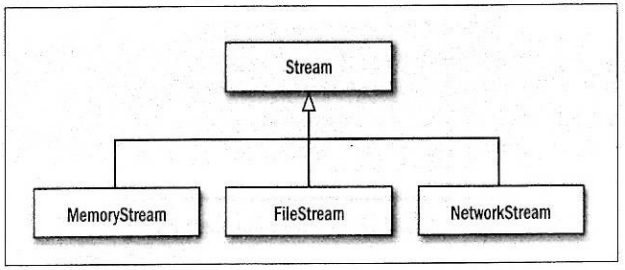


***Figura 1.** El diagrama de clases simplificado de los streams.*

En función de la ubicación en la que se encuentre la información por manipular, deberemos utilizar uno u otro tipo de stream citado. Sin embargo, esto no es todo. La información posee una semántica que deberemos tener en cuenta para saber qué otra clase debemos utilizar en conjunto.

Básicamente, esta semántica consiste en si la información es texto (codificada con algún juego de caracteres) o si es binaria. Entonces, existe otra tipificación de *streams* que se combina con la anterior:

- Lectura de información de tipo texto (clase **StreamReader**).
- Escritura de información de tipo texto (clase **StreamWriter**).
- Lectura de información binaria (clase **BinaryReader**).
- Escritura de información binaria (clase **BinaryWriter**).

Manipulando archivos de texto

Veamos un poco cómo trabajan estas clases. Crearemos una aplicación similar al Notepad que simplemente nos permita abrir y grabar un archivo.
Introduciremos también aquí un nuevo control, que son los diálogos de archivos.

¿QUÉ ES UN STREAM?

Un stream es un objeto que encapsula una secuencia de datos. Los streams de lectura producen información, mientras que los streams de escritura la reciben. Los streams abstraen al programador del modo concreto en el cual se opera con la fuente de datos.

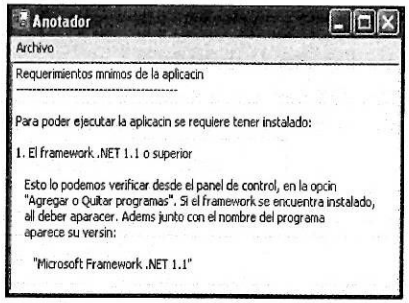


***Figura 2.** Nuestro pequeño anotador.*

Nuestra aplicación poseerá un menú principal, un cuadro de texto (fijado en modo multilínea, y con la propiedad **Dock** fijada en **Fill**), un diálogo de apertura de archivos (clase **OpenFileDialog**) y un diálogo de escritura de archivos (clase **SaveFileDialog**).

Para abrir un archivo a partir de su nombre y copiar su contenido al control **TextBox**, deberemos escribir:

```
StreamReader sr = File.OpenText(nombreArchivo);
txtTexto.Text = sr.ReadToEnd();
sr.Close();
```

En la primera línea hacemos uso del método estático **OpenText** de la clase **File**. Este método nos retornará una referencia a un objeto **StreamReader**, con el cual manipularemos el contenido del archivo.

El objeto **StreamReader** posee los siguientes métodos:

MÉTODO	DESCRIPCIÓN
Close	Cierra el stream.
Peek	Devuelve el siguiente carácter del stream, pero no lo consume (esto significa que en la próxima operación de Peek o Read volveremos a leer el mismo carácter).
Read	Devuelve y consume el siguiente carácter del flujo.
ReadLine	Devuelve un string con la siguiente línea de texto.
ReadToEnd	Devuelve un string con el contenido de texto de todo el stream.

***Tabla 2.** Métodos del objeto SreamReader.*

En el listado de código anterior utilizamos el método **ReadToEnd** porque deseamos colocar **todo** el contenido del archivo en el control (tal vez, si nuestra aplicación trabajase con archivos muy extensos, esto no sería una buena idea).

Finalmente cerramos el *stream* utilizando el método **Close**.

Para grabar el archivo modificado o un archivo nuevo, escribiremos:

```
StreamWriter sw = File.CreateText(nombreArchivo);
sw.Write(txtTexto.Text);
sw.Close();
```

En este caso empleamos el método **CreateText** de la clase **File**, que nos devolverá un objeto del tipo **StreamWriter**.

El objeto **StreamWriter** posee, principalmente, los siguientes métodos:

MÉTODO	DESCRIPCIÓN
Close	Cierra el stream.
Flush	Limpia los buffers relacionados haciendo efectiva la información escrita previamente en el stream.
Write	Escribe información de texto en el stream.

***Tabla 3.** Métodos del objeto StreamWriter.*

Del objeto de tipo **StreamWriter** utilizamos el método **Write** y le pasamos un string que en su interior contiene la información que deseamos almacenar en el archivo. En último término, cerramos el *stream*.

Respecto a los controles de diálogo, su utilización es muy sencilla. Pero cabe destacar que no tenemos por qué utilizarlos; aquí los introducimos simplemente porque es habitual ofrecer este tipo de diálogos al usuario cuando se solicita que se seleccione un archivo para abrir o guardar.

```
dlgAbrirArchivo.Title = "Abrir archivo de texto";
dlgAbrirArchivo.Filter = "Archivos de texto (*.txt)|*.txt|Todos
    los archivos (*.*)|*.*" ;
if (dlgAbrirArchivo.ShowDialog() == DialogResult.OK)
{
    // Almaceno el nombre
    nombreArchivo = dlgAbrirArchivo.FileName;
    // ...
}
```

dlgAbrirArchivo es el objeto de tipo **OpenFileDialog** de nuestro formulario.

En cuanto a la grabación del archivo, el proceso es muy similar. Utilizaremos un objeto del tipo **SaveFileDialog** (llamado **dlgGrabarArchivo**).

```
dlgGrabarArchivo.Title = "Grabar archivo de texto";
dlgGrabarArchivo.Filter = "Archivos de texto (*.txt)|*.txt|Todos
    los archivos (*.*)|*.*" ;
if (dlgGrabarArchivo.ShowDialog() == DialogResult.OK)
{
    nombreArchivo = dlgGrabarArchivo.FileName;
    // ...
}
```

Como se puede apreciar, ambos listados de código son muy parecidos, y es que ambos tipos de diálogo poseen casi las mismas propiedades, ya que son descendientes de la clase **FileDialog**.

Veamos sus propiedades más utilizadas:

PROPIEDADES	DESCRIPCIÓN
FileName	Nombre del archivo seleccionado.
Filter	Especifica las opciones de filtro que se le ofrecerán al usuario para que elija un tipo de archivo válido.
InitialDirectory	Permite especificar en qué directorio comenzará el diálogo.
Title	Título de la ventana de diálogo.

***Tabla 4.** Propiedades de la clase FileDialog.*

Y sus métodos:

MÉTODOS	DESCRIPCIÓN
ShowDialog	Muestra el diálogo en modo modal.

***Tabla 5.** Métodos de la clase FileDialog.*

Manipulando archivos binarios

Crearemos una aplicación que nos permita leer y escribir los valores de las variables de un objeto definido por nosotros arbitrariamente.

Esta vez, de la clase **File** utilizaremos el método **OpenRead** y **OpenWrite**. Ambos métodos nos devolverán un objeto de tipo **FileStream**. A partir de él crearemos otros objetos, con los cuales manipularemos los archivos **BinaryReader** y **BinaryWriter**.

Veamos el código. Leyendo el archivo binario:

```
FileStream fs = File.OpenRead(nombreArchivo);
if (fs != null)
{
    BinaryReader br = new BinaryReader(fs);
    varEntera = br.ReadInt32();
    varString = br.ReadString();
    varFloat = br.ReadSingle();
    br.Close();
}
fs.Close();
```

Como mencionamos, el método **OpenRead** nos devuelve un objeto de tipo **FileStream**. Con él deberemos crear un objeto **BinaryReader**. Luego podremos utilizar los métodos de **BinaryReader** para leer la información del archivo ya abierto.

Entre los métodos más importantes encontramos:

MÉTODOS	DESCRIPCIÓN
Read	Lee un carácter del stream.
ReadBoolean	Lee un valor booleano del stream.
ReadByte	Lee un byte del stream.
ReadBytes	Lee un array de bytes del stream.
ReadChar	Lee un carácter del stream.
ReadChars	Lee un array de caracteres del stream.
ReadDecimal	Lee un número de precisión decimal del stream.
ReadDouble	Lee un número de precisión doble del stream.
ReadInt16	Lee un número entero de 16 bits del stream.
ReadInt32	Lee un número entero de 32 bits del stream.
ReadInt64	Lee un número entero de 64 bits del stream.
ReadSByte	Lee un byte no signado del stream.
ReadSingle	Lee un número de precisión simple del stream.
ReadUInt16	Lee un número entero de 16 bits no signado del stream.
ReadUInt32	Lee un número entero de 32 bits no signado del stream.
ReadUInt64	Lee un número entero de 64 bits no signado del stream.

***Tabla 6.** Métodos del objeto BinaryReader.*

Ahora escribamos el archivo binario.

```
FileStream fs = File.OpenWrite(nombreArchivo);
if (fs != null)
{
    BinaryWriter bw = new BinaryWriter(fs);
    bw.Write(varEntera);
    bw.Write(varString);
    bw.Write(varFloat);
    bw.Close();
}
fs.Close();
```

Para la escritura del archivo binario procedimos de un modo similar. En este caso sólo utilizamos el método **Write** de **BinaryWriter**, no existe la necesidad de especificar diversos métodos con identificadores diferentes, pues el mismo **Write** está sobrecargado para todos los tipos de datos que soporta.
Write: escribe el tipo de dato pasado como parámetro al *stream*. Soporta:

Boolean	Chars	Int32	UInt16
Byte	Decimal	Int64	UInt32
Bytes	Double	SByte	UInt64
Char	Int16	Single	

***Figura 3.** Nuestra pequeña aplicación en acción.*

UTILIZANDO EL NAMESPACE IO

Debido a que todas las clases de manipulación de streams se encuentran bajo el espacio de nombres **IO**, será conveniente que agreguemos la siguiente línea al comienzo de nuestro archivo de código fuente: **using System.IO;**.

BASES DE DATOS

Una base de datos (DB, por su nombre en inglés) es una colección de información relacionada. Físicamente podría estar compuesta por uno o más archivos estructurados internamente para optimizar la manipulación de los datos que contienen.

Hoy en día, muchísimas aplicaciones poseen la necesidad de almacenar sus datos en un medio eficiente, como una base de datos, para su posterior consulta o manipulación. Para esto se podra seleccionar entre muchos productos gestores de bases de datos que existen en el mercado. Los más populares actualmente son:

- **Oracle Database Server**
- **Microsoft SQL Server**
- **Sybase Adaptive Server Enterprise**
- **Informix Dynamic Server**
- **MySQL Database Server**

El **framework .NET** ofrece una serie de clases que facilitan en gran medida la interacción con bases de datos de un modo independiente al fabricante. Estas clases son conocidas como **ADO.NET (ActiveX Data Objects)**.

ADO.NET es una evolución de ADO, y una de sus principales novedades es que ahora permite trabajar de un modo desconectado de la base de datos. Esta decisión ofrece varias ventajas; una de ellas es el ahorro de recursos, ya que en muchas ocasiones las aplicaciones los desperdiciaban permaneciendo extensos períodos de tiempo sin realizar operación alguna (pero con una conexión retenida).

Con ADO.NET nos conectaremos a la base, traeremos un conjunto de datos y finalmente nos desconectaremos de ella. La aplicación trabajará con los datos obtenidos de modo local, y si existe la necesidad de realizar alguna actualización, volverá a conectarse y el ciclo se renovará.

Por otro lado, esta nueva arquitectura es mucho más adecuada para su uso en programación web, por la naturaleza que posee este tipo de aplicaciones.

CONECTIVIDAD A BASES DE DATOS DESDE .NET

Desde .NET podremos conectarnos a casi cualquier base de datos que exista. Si ésta no ofrece clases específicas de ADO.NET, siempre podremos seguir conectándonos a ella mediante ODBC (*Open DataBase Connectivity*).

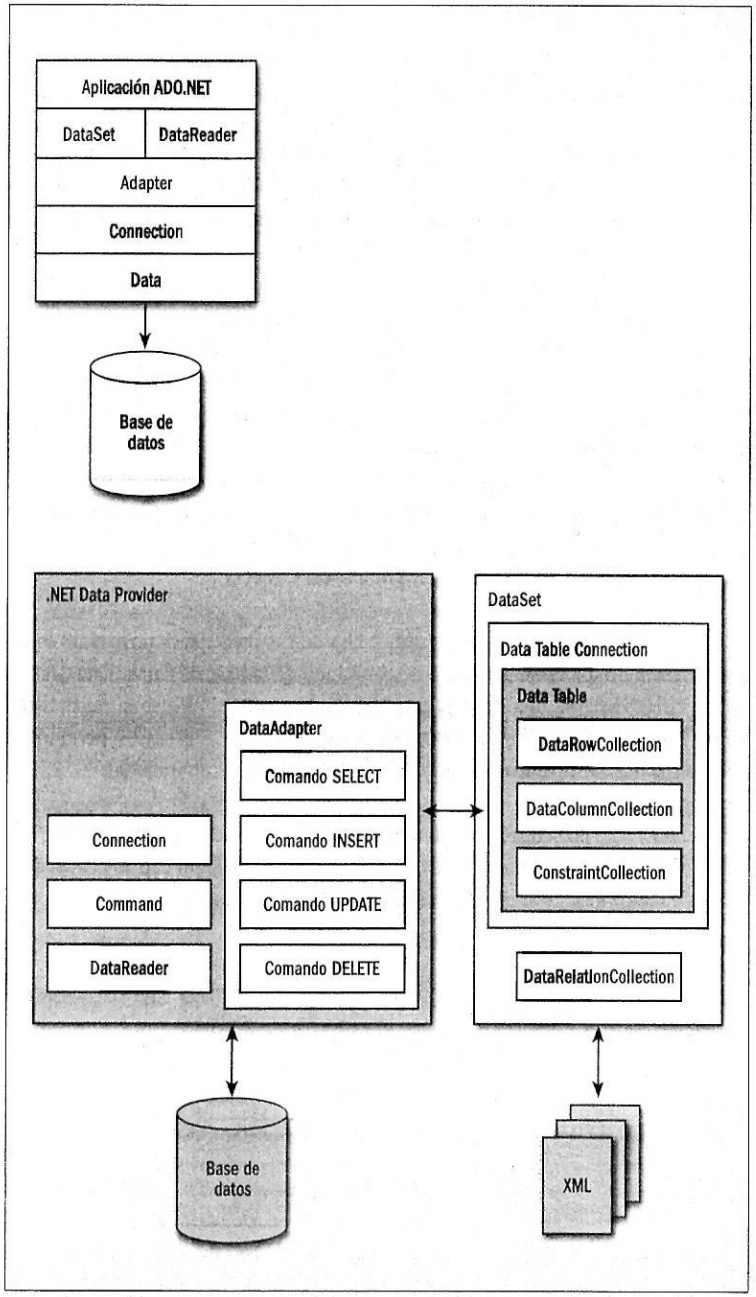


Figura 4. *Esquema de la interacción entre la clase ADO.NET y una base de datos.*

SQL

Podemos encontrar una gran variedad de productos gestores de bases de datos. Afortunadamente para todos nosotros, existe un lenguaje declarativo estándar de consulta llamado **SQL** (*Structured Query Language*).
Con SQL no sólo estaremos posibilitados para consultar la información almacenada en la base de datos, sino que también nos permitirá agregar, modificar y eliminar información, así como modificar las estructuras lógicas donde los datos se encuentran contenidos (por ejemplo, tablas).

No haremos aquí una gran introducción en el lenguaje SQL, sólo nos limitaremos a nombrar los principales comandos y dejaremos al lector la tarea de profundizar en él, en caso de que lo considere conveniente.

Existen tres tipos de comandos SQL:

- **DML** (*Data Manipulation Language*): son los comandos que se encargan de la manipulación de información.

  SELECT: permite consultar por la información almacenada dentro de la base de datos en sus distintos elementos.
  INSERT: permite ingresar información en la base de datos.
  UPDATE: permite modificar la información en la base de datos.
  DELETE: permite eliminar información de la base de datos.

- **DDL** (*Data Definition Language*): comandos para la definición de información.

  CREATE: permite crear los distintos elementos de la base de datos.
  ALTER: permite modificar la estructura de los distintos elementos de la base.
  DROP: permite eliminar los elementos definidos previamente.

- **DCL** (*Data Control Language*): estos comandos están destinados a las tareas de control y configuración de la base de datos.

III ESCRIBIENDO COMANDOS SQL

Conocer SQL para trabajar con bases de datos es muy recomendable. Sin embargo, si no nos manejamos muy bien con este lenguaje, podremos utilizar los asistentes que ofrecen los controles de ADO.NET para que sean ellos los que construyan los comandos SQL por nosotros.

DENY: niega permisos a usuarios para realizar determinadas tareas.
GRANT: otorga permisos a usuarios para realizar determinadas tareas.
REVOKE: revoca permisos otorgados.

Elementos de una base de datos relacional

Una base de datos se encuentra compuesta por diversos elementos. Entre los más destacables podemos mencionar:

- **Tabla**: es el elemento fundamental. Una tabla representa una entidad y sus relaciones; contiene unidades de información denominadas **registros**, que especifican valores para cada campo que la tabla puede almacenar (determinados en **columnas**). Posee un nombre mediante el cual se referencia.

CÓDIGO	NOMBRE	APELLIDO	EDAD
1	juan	ritz	20
2	pedro	pua	23
3	carlos	toms	25
4	ivan	werty	26

Columnas
Registros

***Figura 5.** Una tabla posee columnas y registros.*

- **Índices**: los índices son estructuras de datos que se relacionan con una o más columnas de una tabla, con el fin de acelerar el proceso de búsqueda de registros cuando se utilizan los campos de dichas columnas como criterio.
- **Vistas**: son tablas virtuales creadas dinámicamente a partir de consultas preestablecidas. Poseen la virtud de simplificar el acceso a la información.
- **Procedimientos almacenados**: son scripts administrados por el gestor de bases de datos que actúan, principalmente, sobre los elementos de ésta. El lenguaje en el cual se programan estos scripts dependerá del gestor de la base de datos.
- **Disparadores (triggers)**: son scripts que se disparan cuando se suceden ciertas operaciones sobre los elementos de la base de datos (por ejemplo: cuando se intenta agregar un registro a una tabla).

Los elementos de una base de datos pueden ser creados, modificados o eliminados por medio de comandos SQL (del tipo **DDL**). Sin embargo, es bastante frecuente emplear aplicaciones más amistosas que nos faciliten la tarea mediante una interfaz de usuario (estas aplicaciones se encargan de generar los comandos SQL que correspondan a partir de nuestra interacción).
ADO.NET ofrece un poderoso conjunto de clases para manipular la información de la base de datos. Ejecutar comandos SQL es sólo una parte ínfima de sus funcionalidades. Veamos cuáles son las clases más importantes.

DataSet

Un **DataSet** es un conjunto de datos provenientes de una base de datos, de un archivo **XML**, creados dinámicamente con código, etc.

En el caso habitual contendrá una o más tablas relacionadas que serán obtenidas de una fuente. Podremos trabajar con el **DataSet** consultando por la información que contiene, agregando, modificando o eliminando su contenido.

Posteriormente, si hemos realizado cambios, y en el caso de que deseemos que estas modificaciones se hagan efectivas, podremos recorrer el camino inverso y actualizar la fuente de datos con el **DataSet** (sólo cuando la fuente lo permita).

Internamente, un **DataSet** posee:

- Una colección de tablas (tipo **DataTable**).
- Una colección de registros por tabla.
- Una colección de columnas por tabla.
- Una colección de restricciones por tabla.
- Una colección de relaciones.

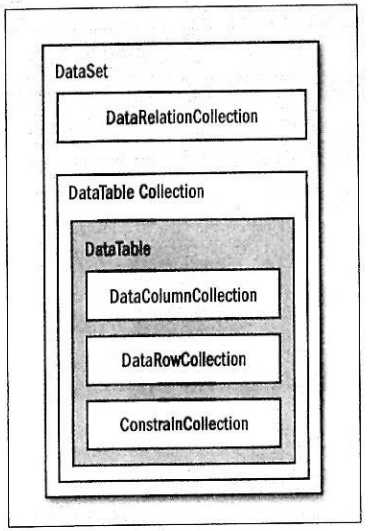


***Figura 6.** El interior de un DataSet.*

Las tablas no tienen por qué estar colocadas dentro del **DataSet** de modo independiente, sino que también pueden poseer relaciones que se expresan a través de objetos que se encuentren dentro de él.

Connection

Mediante un objeto del tipo **Connection** se realizará la conexión a la base de datos.

DataAdapter

Hemos mencionado anteriormente que un **DataSet** se alimenta de una fuente de datos, que en este momento para nosotros sería una base de datos.
El modo en el cual el **DataSet** se conecte a dicha base será por medio de un objeto del tipo **DataAdapter**, que se encarga de hacer de puente para la información, ya sea desde la base hacia el **DataSet** o por el camino inverso.

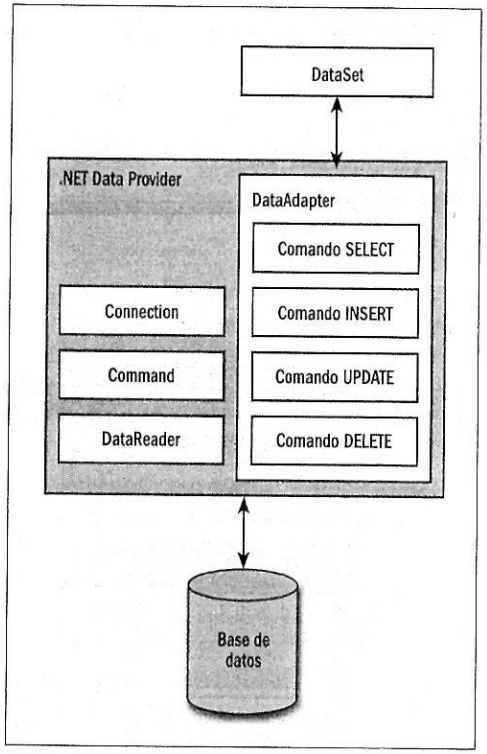


***Figura 7.** Un DataSet se conecta a una base de datos a través de un DataAdapter.*

Internamente, un **DataAdapter** estará compuesto por comandos SQL, que serán utilizados para las operaciones que el **DataSet** requiera realizar con la base.

Command

Representa un comando SQL, el cual se ejecuta a una fuente de datos utilizando para tal propósito un objeto del tipo **DataAdapter**.

DataReader

Permite navegar hacia adelante por registros de una base de datos mientras se está conectado a ella. Una aplicación suele optar por utilizar **DataReaders**, en lugar de **DataSets**, cuando la cantidad de datos por traer es muy grande (con **DataSets** se utiliza mucha memoria) y se requiere un acceso de consulta muy rápido.

UN EJEMPLO SENCILLO

Ahora que hemos introducido los principales conceptos de ADO.NET, podremos ver un sencillo ejemplo. En nuestro caso utilizaremos una base de datos de tipo Microsoft Access. Ésta contendrá sólo una tabla que manipularemos utilizando C# y las clases ADO.NET.

Nuestro primer paso será crear una pequeña base de datos con una tabla sencilla, que luego consultaremos desde una aplicación C#.

ID	NOMBRE	EDAD	CALLE	PISO	DEPTO	TELÉFONO
1	usuario1	25	CalleX	1	A	4555-0001
2	usuario2	26	CalleY	1	C	4555-0009
3	usuario3	23	CalleZ	4	B	4555-0201
4	usuario4	22	CalleW	3	C	4555-0441

***Figura 8.** Nuestra tabla de **Usuarios**.*

Con la tabla creada veremos ahora cómo operar desde **Visual Studio**. Para esto construiremos una aplicación con interfaz de usuario del modo tradicional y arrojaremos un control **DataGrid** al formulario.

El control **DataGrid** es bastante práctico para interactuar con **DataSets**. Una vez que éste se encuentre colocado en la porción del formulario que más nos agrade, agregaremos otros controles más. En este caso, dentro del **Toolbox**, cambiaremos de

¿QUÉ BENEFICIOS DA UTILIZAR UNA BASE DE DATOS?

Utilizar una base de datos –en vez de archivos binarios– trae muchísimas ventajas debido a que son sistemas creados para organizar y gestionar información. Entre otras características, podremos contar con integridad referencial de datos, seguridad de acceso, optimización de consultas, etc. El uso de archivos binarios se justifica sólo en aplicaciones muy sencillas.

grupo y pasaremos de **WebForms** a **Data**. Dentro de esta sección podremos encontrar básicamente tres tipos de controles:

- **Connection**
- **DataAdapter**
- **Command**

Cada uno de ellos poseerá un prefijo:

- **OleDb**: conexión a una fuente de información a través de un proveedor de datos de tipo **OLE DB**. Este tipo de conexión será la que utilizaremos para trabajar con bases de datos creadas por la aplicación **Microsoft Access (Jet DB)**.
- **Sql**: conexión a bases de datos **Microsoft SQL Server**.
- **Oracle**: conexión a bases de datos **Oracle**.
- **Odbc**: conexión a una fuente de información a través de un proveedor de datos ODBC (*Open DataBase Connectivity*).

Todos los objetos de tipo **Connection** implementan la interfaz **IDbConnection**. Esto es muy positivo, ya que cada implementación tendrá las particularidades que requiera para dialogar con el producto específico para el cual fue construida, pero, al mismo tiempo, tendrá el mismo conjunto de propiedades y funciones.

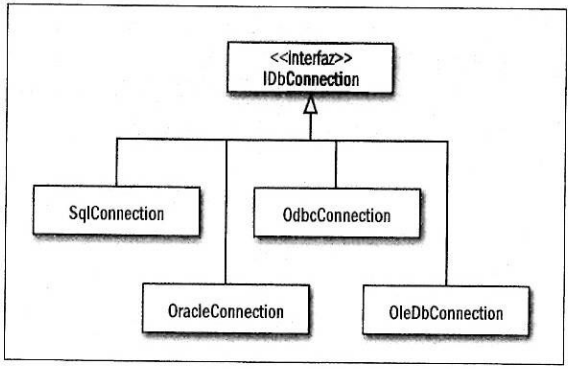


***Figura 9.** Clases que implementan la interfaz IDbConnection.*

III ALTERNATIVAS DE CONEXIÓN

Es posible que, en ciertas bases de datos, tengamos varias alternativas para conectarnos a ellas. En el caso de Oracle, por ejemplo, podremos utilizar OracleConnection y OdbcConnection. Siempre será mejor usar el objeto específico creado para dicha base, debido a que posee optimizaciones propias del fabricante.

De la misma manera que en el caso anterior, los objetos del tipo **Command** implementan la interfaz **IDbCommand**.

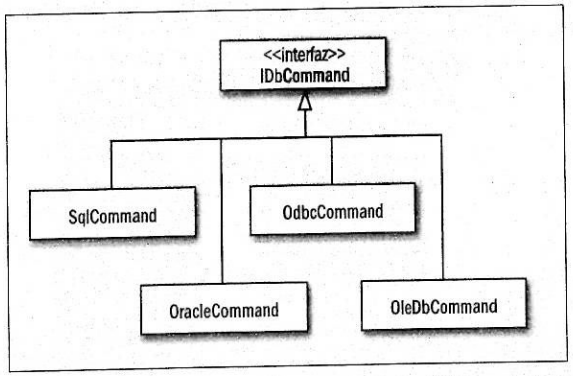


***Figura 10.** Clases que implementan la interfaz IDbCommand.*

Y los objetos del tipo **DataAdapter** implementan la interfaz **IDbDataAdapter**.

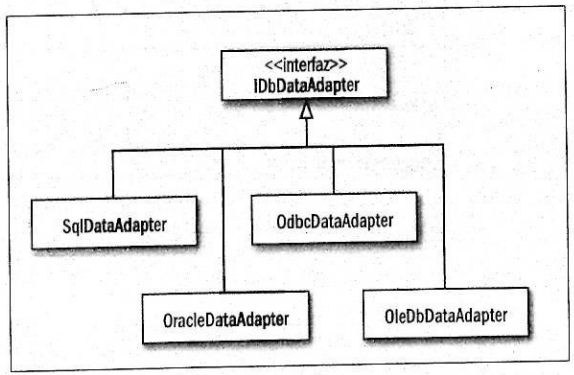


***Figura 11.** Clases que implementan la interfaz IDbDataAdapter.*

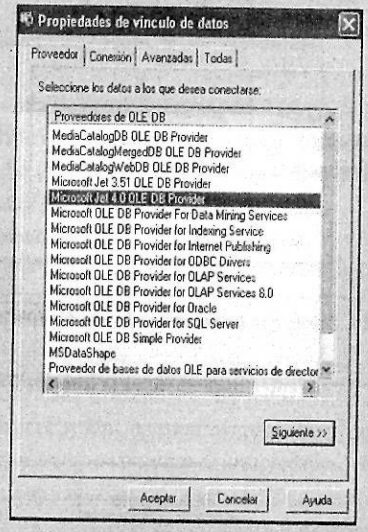
Si bien aquí veremos cómo utilizar objetos del tipo **OleDb**, en principio no habrá grandes diferencias en el uso de objetos para otros proveedores.

Por lo tanto, tomamos un control del tipo **OleDbConnection** y se lo agregamos a nuestro formulario. De este control será importante especificar la propiedad **ConnectionString**, que será la que se ocupe de indicar al objeto la ubicación y el modo de conectarse a la base de datos.

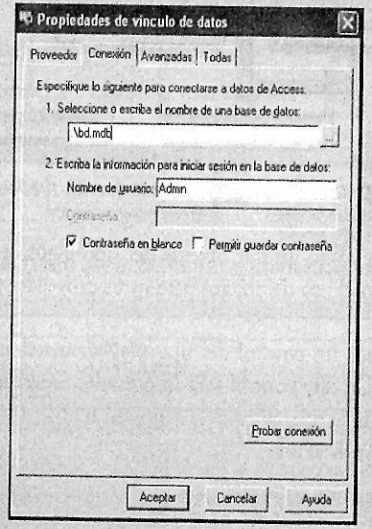
Esta propiedad puede ser estipulada por código o bien por medio de un asistente que se dispone con esta finalidad.

Asistente para conexión a la base de datos PASO A PASO

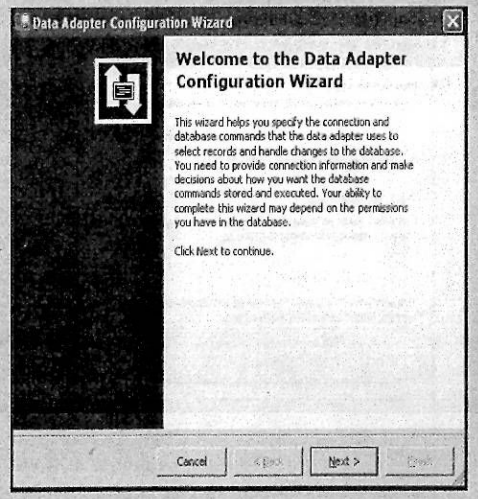
1 Si utilizamos el asistente, deberemos seleccionar **Microsoft Jet 4.0 OLE DB Provider** como proveedor de datos.



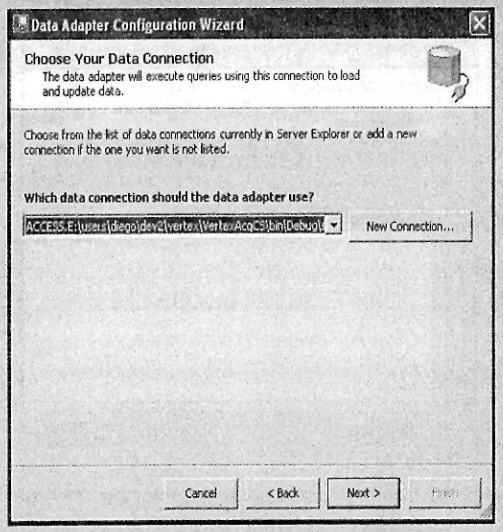
2 Luego, en **Conexión** podremos indicar la ruta completa a la base de datos.



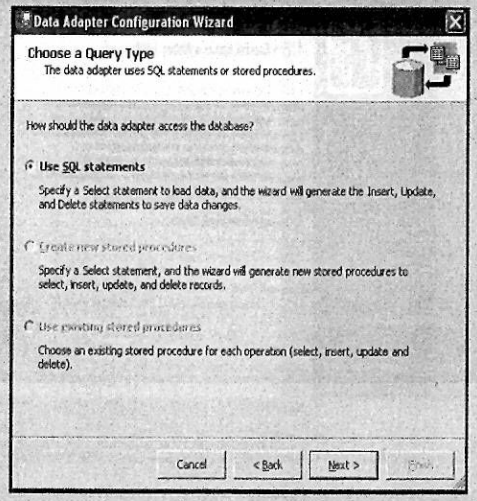
3 Ahora, agregaremos el control **OleDbDataAdapter**. Cuando hagamos esto, un asistente se abrirá automáticamente y nos ofrecerá sus servicios.



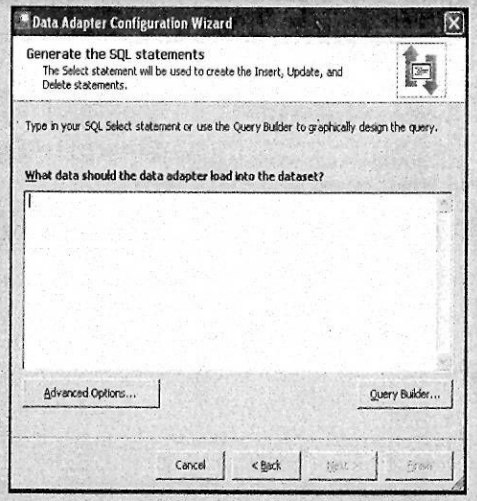
4 Aquí deberemos seleccionar el objeto de tipo **Connection** por el cual nos conectaremos a la base de datos (en nuestro caso será el que hemos creado en las líneas anteriores).



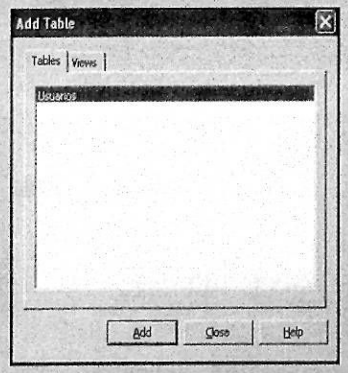
5 Especificamos el modo de acceder a la base de datos. Las opciones que se encontrarán disponibles dependerán en forma exclusiva de las funcionalidades que ofrece el gestor de base de datos.



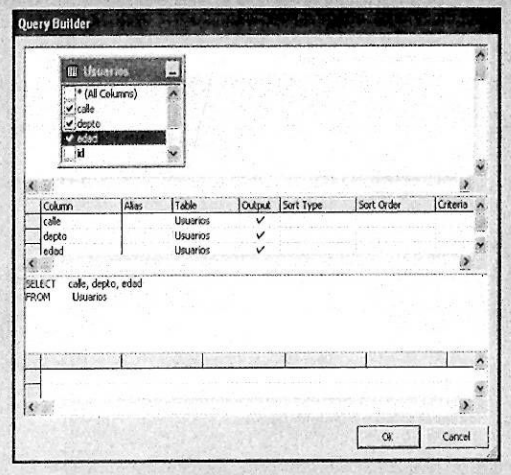
6 Ahora deberemos especificar el comando **SELECT**, por medio del cual se traerá la información desde la base de datos a través del **DataAdapter**. Si no sabemos **SQL**, podremos utilizar un creador de consultas presionando del botón **Query Builder**.



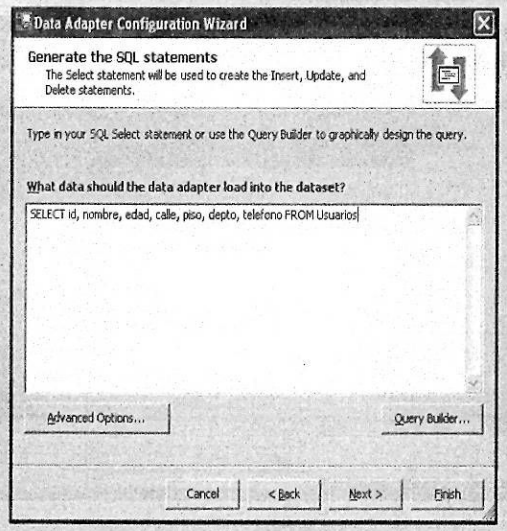
7 La opción **Query Builder** se encargará de solicitarnos que seleccionemos la totalidad de las tablas que se utilizarán para realizar nuestra consulta (en nuestro ejemplo, sólo disponemos de una).



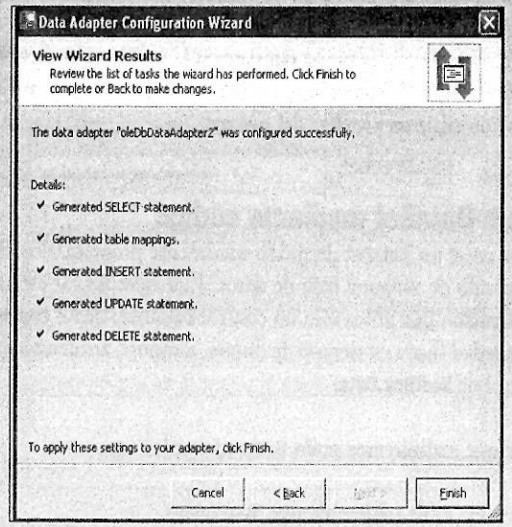
8 Aún en el **Query Builder**, luego de que hayamos seleccionado las tablas, también deberemos indicar cuáles serán las columnas que formarán parte de la consulta (no nos encontramos en la obligación de traer todas las columnas de una tabla, sino solamente las que vayamos a utilizar).



9 Una vez que hemos cerrado el **Query Builder**, éste nos deja el comando **SELECT** escrito en nuestro código.



10 En el último paso, el asistente nos informa los comandos que fueron creados por nosotros para manipular la información seleccionada.



Bien. Tenemos nuestro **OleDbConnection**, que nos conectará a la base de datos, y nuestro **OleDbDataAdapter**, que se encargará de hacer de puente para traer los datos

de la base. Pero ¿dónde los alojará? Podríamos especificar para ello un **DataSet** programáticamente o en tiempo de diseño.

Para especificarlo en tiempo de diseño deberemos seleccionar el **DataAdapter**, acceder a su menú contextual y luego seleccionar la opción **Generate Dataset....**

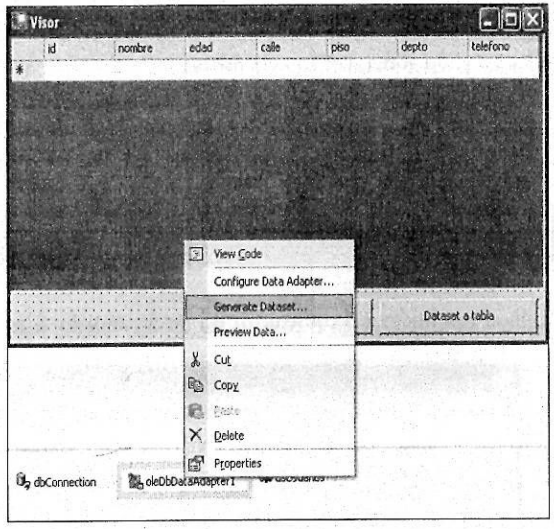


***Figura 12.** Generando el DataSet.*

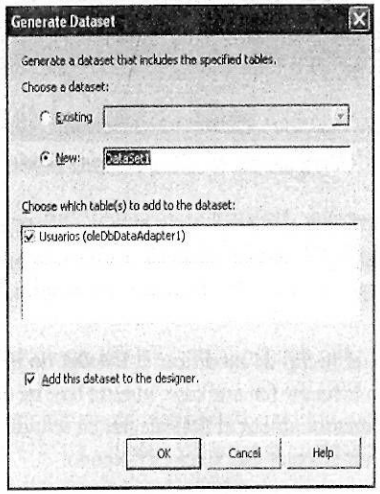


***Figura 13.** Seleccionando el nombre de nuestro DataSet tipado.*

Completada esta operación, notaremos que nuestro proyecto posee una nueva clase. Esta clase es un **DataSet** tipado, creado específicamente para nuestras tablas. Esto quiere decir que ahora podremos manipular las columnas, los registros, las restricciones y las relaciones de nuestro objeto por medio de propiedades y métodos, como si fuese un objeto más.

Veamos ahora cómo, casi sin escribir código, podremos utilizar el control **DataGrid** para manipular nuestro **DataSet** llamado **dsUsuarios**.
Para esto simplemente deberemos fijar la propiedad **DataSource** del control **DataGrid** a nuestro **DataSet**. Luego, mediante código indicaremos de modo explícito que se consulte la información de la base de datos:

```
adpUsuarios.Fill(dsUsuarios);
```

Podríamos colocar esta línea en el constructor del formulario, si así lo deseáramos.

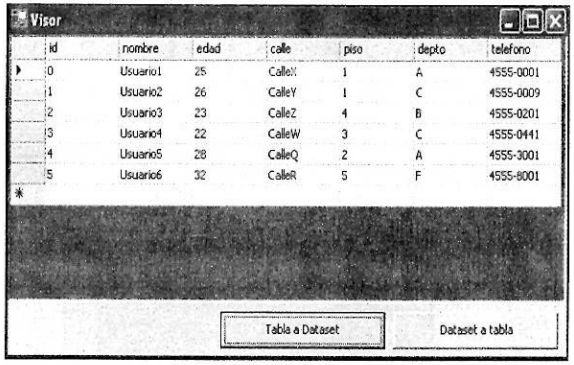


***Figura 14.** El control DataGrid en acción.*

Por predefinición, mediante el control podremos modificar la información contenida dentro del **DataSet** (o de la tabla seleccionada del **DataSet**). Este comportamiento puede ser modificado si fijamos la propiedad **ReadOnly** en verdadero.

De todos modos, el hecho de modificar el **DataSet** no implica que los cambios se hagan efectivos en la fuente (en este caso, nuestra base de datos). Para hacer los cambios efectivos deberemos utilizar el **DataAdapter** en sentido inverso (conceptualmente, claro). Podremos efectuar esta tarea escribiendo:

```
adpUsuarios.Update(dsUsuarios, "Usuarios");
```


Agregando datos al DataSet

Para agregar datos al **DataSet** de manera programática, deberemos proceder del siguiente modo:

```
// Creo un nuevo registro
DataSetUsuarios.UsuariosRow newrow = dsUsuarios.Usuarios.NewUsuariosRow();
// Especifico sus propiedades (columnas de la tabla del dataset)
newrow.id = 10;
newrow.nombre = "guadalupe";
// Agrego el registro nuevo
dataTable.AddUsuariosRow(newrow);
// Actualizo la tabla (dataset -> tabla)
adpUsuarios.Update(dataSet11, "Usuarios");
```

Analicemos el listado anterior.

En la primera línea de código creamos un nuevo registro del tipo **UsuariosRow**. Este tipo de dato fue creado automáticamente en el momento en que generamos el **DataSet** mediante el asistente citado.

Como habíamos mencionado previamente, la ventaja de poseer un **DataSet** tipado es enorme. Una de esas ventajas consiste en la verificación de tipos de datos que se realizará en tiempo de compilación evitando de esta manera que cualquier error de programación surja en pruebas del sistema.

Crear un DataSet mediante código

Es posible crear un **DataSet** de modo totalmente programático sin que su contenido sea extraído de ninguna base de datos. Para ejemplificar esto crearemos una pequeña aplicación que posea sólo un control **DataGrid**; dicho control no poseerá ninguna propiedad fijada en tiempo de diseño, tampoco arrojaremos al formulario ningún control de **ToolBox.Data**.

Los pasos que realizaremos serán básicamente los siguientes:

1.a. Crear una tabla (objeto de tipo **DataTable**).
1.b. Especificar las columnas de dicha tabla.
1.c. Crear un **DataSet** (objeto de tipo **DataSet**).
1.d. Agregar la tabla creada al **DataSet** anterior.

2.a. Crear un nuevo registro.
2.b. Especificar el valor de los campos de dicho registro.
2.c. Agregar el nuevo registro a la tabla de nuestro **DataSet**.

3. Relacionar el **DataSet** al control **DataGrid**.

Veamos el código para el paso 1:

```
// 1.a. Creamos una tabla
DataTable tbArchivos = new DataTable("Archivos");
// 1.b. Agrego dos columnas
tbArchivos.Columns.Add("nombre", Type.GetType("System.String"));
tbArchivos.Columns.Add("tamaño", Type.GetType("System.Int32"));
// 1.c. Creamos el dataset
dsArchivos = new DataSet();
// 1.d. Agregamos la tabla al dataset
dsArchivos.Tables.Add(tbArchivos);
```

Del paso 2:

```
DirectoryInfo di = new DirectoryInfo("c:\\");
// Get a reference to each file in that directory.
FileInfo[] fiArr = di.GetFiles();
// Display the names of the files.
foreach (FileInfo fri in fiArr)
{
    // 2.a. Creo un registro
    DataRow dw = dsArchivos.Tables["Archivos"].NewRow();
    // 2.b. Especifico valores para el registro
    dw["nombre"] = fri.Name;
    dw["tamaño"] = fri.Length;
    // 2.c. Agrego el registro nuevo a la tabla
    dsArchivos.Tables["Archivos"].Rows.Add(dw);
}
```

Préstese especial atención a que los registros se encuentran compuestos por información de consulta de archivos a un directorio arbitrario (esta tarea es simplemente para tomar datos de algún lugar).

Y, finalmente, el paso 3:

```
dgDirs.DataSource = dsArchivos.Tables["Archivos"];
```

En este último caso, el control también nos permite especificar, no una tabla, sino un **DataSet** completo. En dicho caso ofrece un enlace con los nombres de las tablas que componen el **DataSet** para ingresar en ella.

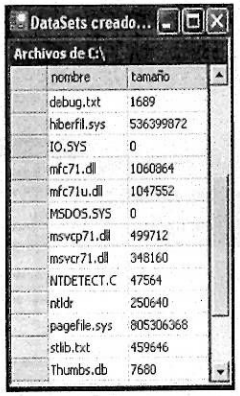


***Figura 15.** Nuestro ejemplo en acción.*

RESUMEN

En el mundo real, las aplicaciones interactúan con datos externos a la aplicación misma: archivos, bases de datos, objetos remotos, etc. En este capítulo hemos introducido varios conceptos importantes y, en particular, ADO.NET, un tema del cual podríamos escribir un libro entero. En el siguiente capítulo estudiaremos cómo manejar los errores que podrían producirse en nuestros programas por medio de la manipulación de excepciones.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

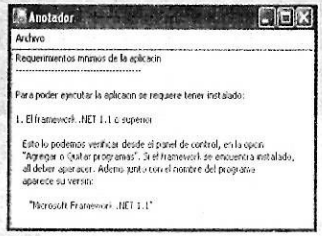
1 ¿Por qué la clase BinaryReader no posee sólo un método del tipo Read sobrecargado devolviendo en cada caso el tipo de dato que corresponda?

2 ¿Podría un DataSet poseer información de más de una base de datos?

3 ¿Qué es un DataSet tipado? ¿Qué beneficios otorga? ¿Es posible interactuar con una base de datos con un DataSet no tipado?

EJERCICIOS PRÁCTICOS

✔ Modificar la aplicación Anotador (que lee y escribe archivos de texto) para que se permita la modificación de la tipografía utilizada por el control TextBox (haciendo uso del control FontDialog). Agregar, además, un formulario con información del tipo Acerca de....



Capítulo 9

Manejo de excepciones

¿Qué sucede cuando en la ejecución de nuestros programas las cosas no salen de la manera prevista? ¿Qué ocurre cuando no es posible encontrar aquel archivo que necesitamos abrir o cuando el usuario ingresa un dato inválido? C# ofrece un sistema de manejo de excepciones muy completo que nos permitirá lidiar con estas situaciones de un modo muy sencillo y efectivo.

SERVICIO DE ATENCIÓN AL LECTOR: lectores@mpediciones.com

TRATAMIENTO DE ERRORES

Sabemos que los programas que escribimos pueden poseer errores, algunos de los cuales son detectados en tiempo de compilación (errores de sintaxis, semánticos, etc.), pero otros podrían permanecer inadvertidos por ser errores de lógica o requerir ciertas condiciones para manifestarse.

Si debemos efectuar una división y el denominador es provisto por una variable ingresada por el usuario, el proceso de compilación finalizará de manera satisfactoria. Pero en nuestro programa podría producirse un error en el supuesto caso de que el usuario especificara cero como denominador.

Por otro lado, siempre existe la posibilidad de que se presenten situaciones imprevistas, a pesar de no que no existan errores de programación en nuestra aplicación (siempre y cuando no tomar todas las precauciones en cada caso no sea considerado por nosotros como un tipo de error).

C# ofrece un sistema para manejar este tipo de situaciones, heredado del lenguaje C++, que se denomina "manejo de excepciones" (*exception handling*).

Una excepción es una alteración en el flujo de ejecución normal de la aplicación. El sistema de manejo de excepciones tiene como principal finalidad permitir el tratamiento de errores en nuestros programas.

Hasta el momento, el código desarrollado por nosotros no ha hecho tratamiento de excepciones. Por ejemplo, la línea de código que observamos a continuación parecería no poseer inconveniente alguno:

```
int num = Convert.ToInt32(txtNum.Text);
```

Aquí, nuestra intención fue convertir a número entero el contenido de un cuadro de texto especificado.

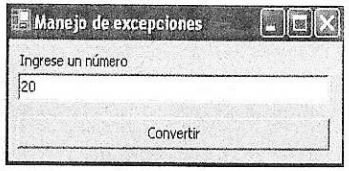


***Figura 1.** Convirtiendo texto a número.*

Cuando el usuario ingrese un número dentro del cuadro de texto, la función de conversión retornará con el resultado de modo satisfactorio. Pero ¿qué sucedería si el cuadro de texto posee algo que no sea un número en su interior? La conversión fallará, y en pantalla observaremos la presencia de un mensaje que nos informará de lo ocurrido, luego de lo cual nuestra aplicación se cerrará.

Si estamos ejecutando la aplicación desde el entorno de desarrollo, veremos en pantalla el siguiente mensaje, y luego se marcará la línea que produjo la ocurrencia.

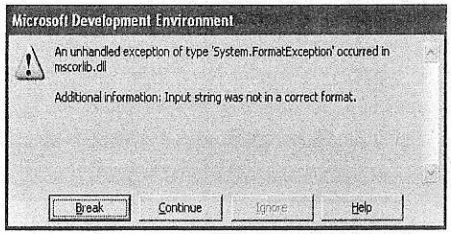


***Figura 2.** ¡Oops! Algo salió mal.*

Si, en cambio, estamos ejecutando la aplicación externamente, veremos:

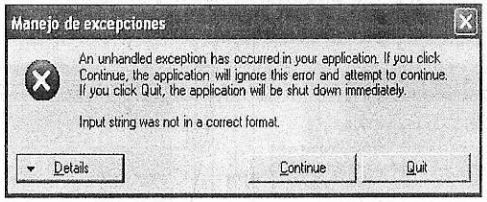


***Figura 3.** Éstas no son buenas noticias.*

Lo que ocurrió fue que el método estático **ToInt32** de la clase **Convert** arrojó una excepción. Éste es un modo de avisarnos que algo salió mal, por lo tanto deberíamos hacer algo al respecto o la aplicación no podrá seguir su curso normal.

III EXCEPCIONES DE LA LIBRERÍA BCL

Las clases de la librería BCL arrojarán excepciones cuando los parámetros enviados no sean los esperados o cuando se produzca algún tipo de evento no esperado. Siempre que algún método de estas clases "pueda fallar", será conveniente encerrarlo en un bloque **try..catch**.

ENCERRAR LAS EXCEPCIONES

Es posible capturar las excepciones que puedan arrojar los diversos métodos que invoquemos. Para esto deberíamos encerrar el código que podría arrojar la excepción en un bloque **try..catch**. Veamos:

```
try
{
    num = Convert.ToInt32(txtNum.Text);
}
catch
{
    MessageBox.Show("Se ha producido una excepción");
}
```

Éste es el modo más básico de manejar una excepción. Ahora, si se llega a producir alguna dentro del código del bloque **try**, el flujo de ejecución saltará automáticamente a la primera línea del bloque **catch**. Luego, la ejecución del programa continuará normalmente, si es que no establecemos lo contrario.

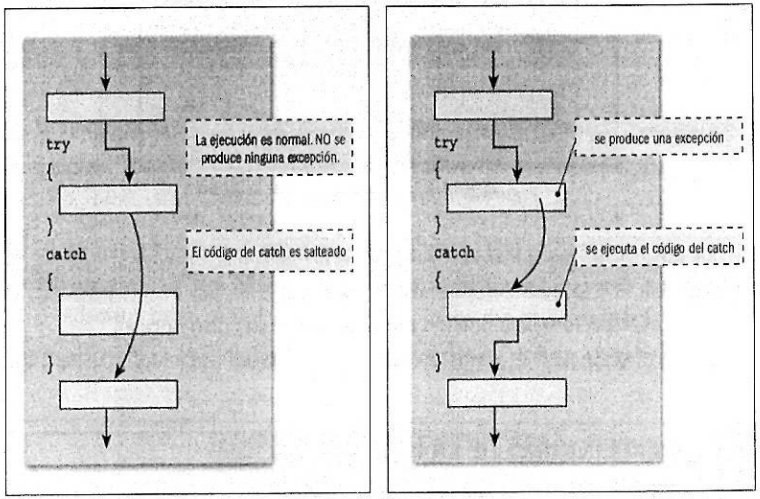


Figura 4. *Izquierda: no se produce excepción. Derecha: se produce una excepción, por lo que se ejecuta el código del catch.*

Es cierto que también podríamos realizar las verificaciones del caso para evitar que pueda producirse una excepción. Es decir que en el caso anterior podríamos haber

verificado que el texto por convertir contuviera sólo números. Sin embargo, estas verificaciones terminan agregando código que siempre será ejecutado y no nos asegura que, de todos modos, se produzca la excepción por alguna verificación que hemos olvidado o que ni siquiera pueda realizarse.
Además, de este modo se mezclaría el código relacionado con la lógica del programa con el código asociado al manejo de errores. En cambio, utilizando un bloque **try..catch** se diferencia a simple vista cuál es el objetivo de cada trozo de código.

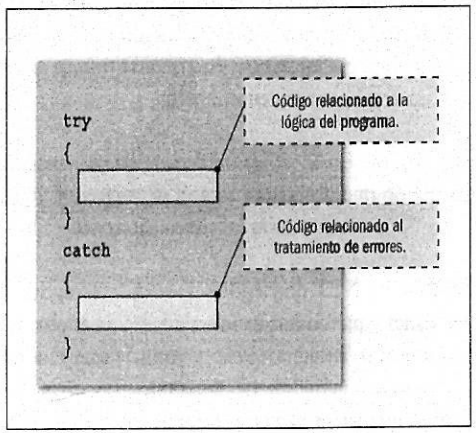


Figura 5. *Podemos diferenciar bien qué hace cada trozo de código.*

CLASES DE EXCEPCIONES

Hemos logrado capturar una excepción arrojada por un método, encerrando su invocación dentro de un bloque **try..catch**.
Pero ¿qué fue lo que ocurrió? ¿Qué produjo el error? ¿Qué le informamos al usuario? Existe más información que podremos obtener al respecto; el bloque **catch** puede recibir un objeto de algún tipo de excepción.

Rehagamos nuestro ejemplo y veamos que podremos ofrecerle al usuario no sólo la indicación de que algo ha fallado.

```
try
{
    num = Convert.ToInt32(txtNum.Text);
}
```

```
catch (FormatException fe)
{
    MessageBox.Show(fe.Message);
}
```

Ahora, si se produce una excepción, en el cuadro de texto no habrá un mensaje genérico, sino información un poco más específica de lo que ocurrió:

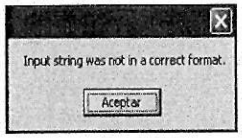


***Figura 6.** Enviando a un cuadro de mensaje información de la excepción capturada.*

Las clases de excepción en **.NET** descienden de la clase **Exception**, la cual ofrece un conjunto de propiedades y métodos que nos ayudarán en la obtención de mayor información y en un mejor tratamiento de la ocurrencia que generó la excepción. Existen muchas clases descendientes de **Exception**. En el siguiente diagrama se encuentran detalladas algunas de ellas:

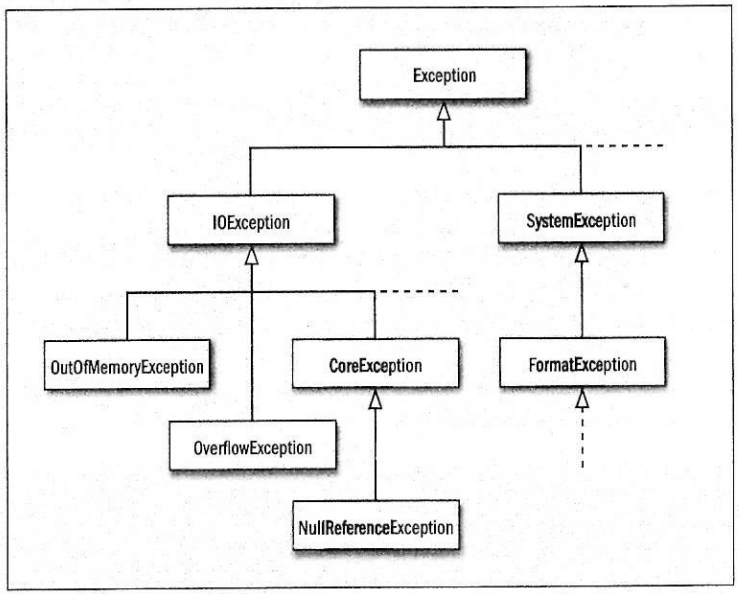


***Figura 7.** El diagrama de excepciones del framework .NET.*

En caso de que se produzca una excepción del tipo **FormatExeption**, podremos capturarla especificando como parámetro un objeto del tipo **FormatExeption** o un objeto de tipo superior (como, por ejemplo, el objeto **SystemException**).

Por lo tanto, el siguiente código también será válido:

```
try
{
    num = Convert.ToInt32(txtNum.Text);
}
catch (SystemException se)
{
    MessageBox.Show(se.Message);
}
```

o incluso:

```
try
{
    num = Convert.ToInt32(txtNum.Text);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

La idea de este sistema cobra sentido cuando comprendemos que pueden existir múltiples bloques **catch** por cada bloque **try**, y el flujo de ejecución se desviará al que corresponda según la excepción generada.

CONOCER EL TIPO DE EXCEPCIÓN

Los nombres de las clases de excepción predefinidas son muy elocuentes, por lo que podremos intuir qué tipo de excepción es arrojado en cada caso. Luego podremos consultar la documentación de la librería **BCL** (incluida con el **framework SDK** y **Visual Studio .NET**). Lo difícil será entender lo que nos arroje el **framework .NET** cuando la excepción se produzca por primera vez.

Supongamos el siguiente ejemplo:

```
int num, res;
try
{
    num = Convert.ToInt32(txtNum.Text);
    res = 5 / num;
}
catch (FormatException fe)
{
    // Tratamiento de la excepción de formato
}
```

Si se produce una excepción de formato (había texto dentro de **txtNum.Text**), ésta será tratada por el bloque **catch** que especificamos. Pero ¿qué ocurre si **num** es igual a cero? También se producirá una excepción, pero no de tipo formato, sino de tipo **división por cero**, y nuestro bloque **catch** no la aceptará, por lo que estaremos como en el caso inicial (sin tratar excepciones) y la ejecución de nuestra aplicación se detendrá.

Entonces, estamos ante dos posibles soluciones:

- Especificamos una clase más general en el tipo de objeto que recibe **catch** para que ésta abarque los dos casos.
- Creamos otro bloque **catch** con la clase de excepción que corresponda.

La primera parece buena idea, pero parte de la gracia aquí está en poder especificar diferente código para cada una de las ocurrencias que se podrían suscitar ante un error, es decir que si existiera una excepción de formato, querríamos ejecutar cierto código, y si existiera una excepción de división por cero, querríamos ejecutar otro.

Entonces veamos cómo implementar nuestra segunda opción:

III IDENTIFICADORES DE OBJETOS DE EXCEPCIÓN

El identificador con el cual declaremos el objeto de excepción es totalmente arbitrario. En los ejemplos usamos **se** o **ex** simplemente porque son cortos. De todos modos, dicho objeto sólo será válido dentro del bloque **catch** para el cual fue declarado, y no deberían ser muy extensos en cantidad de líneas, por lo que no existe la necesidad de crear un identificador superexpresivo.


```
int num, res;
try
{
    num = Convert.ToInt32(txtNum.Text);
    res = 5 / num;
}
catch (FormatException fe)
{
    // Tratamiento de la excepción de formato
}
catch (DivideByZeroException dbze)
{
    // Tratamiento de la excepción división por cero
}
```

Ahora, en función de lo que ocurra, el flujo de ejecución se dirigirá a un lugar o a otro.

Podremos especificar tantas ramas **catch** como tipos de excepciones queramos capturar; con la única restricción de no poder especificar un **catch** con una clase de excepción que sea base de la clase de excepción de otro **catch** que se encuentre debajo de él. Expresado de otro modo, sabemos que **FormatException** desciende de **Exception**. Siempre deberemos ir en un sentido **de más específico a más general**.

Por lo tanto, el siguiente código no es válido:

```
int num, res;
try
{
    num = Convert.ToInt32(txtNum.Text);
    res = 5 / num;
}
catch (Exception ex)
{
    // Tratamiento de la excepción
}
catch (FormatException fe)          // ¡Error!
{
    // Tratamiento de la excepción de formato
}
```

En cambio, el siguiente sí lo es:

```
int num, res;
try
{
     num = Convert.ToInt32(txtNum.Text);
     res = 5 / num;
}
catch (FormatException fe)
{
     // Tratamiento de la excepción de formato
}
catch (Exception ex)
{
     // Tratamiento de la excepción
}
```

También podremos especificar un bloque **finally**, el cual se ejecutará siempre, ocurra o no ocurra excepción.

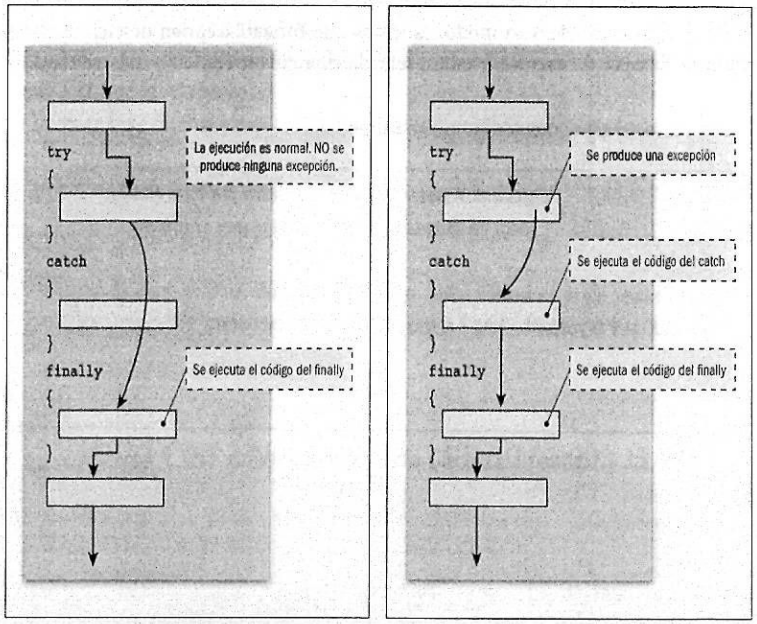


***Figura 8.** El código dentro del bloque **finally** se ejecuta siempre.*

En el bloque **finally** se suele colocar código relacionado con la liberación de recursos adquiridos posterior a la ocurrencia de una excepción.

Por ejemplo, si se ha abierto un archivo dentro del bloque **try**, es conveniente cerrarlo, se haya producido una excepción o no.

Veamos esto en pseudocódigo:

```
try
{
     Abrir archivo
     ...
}
catch (AlgunaExcepción e1)
{
     Tratamiento de excepción e1
}
catch (OtraExcepción e2)
{
     Tratamiento de excepción e2
}
finally
{
     Cerrar archivo
}
```

De este modo, si se suscita una excepción o no, igual se ejecutará el código correspondiente al **finally**.
Claro que usted podrá pensar: pero... ¿podría cerrar el archivo luego de todo el bloque **try..catch**? Para comprender la respuesta a este interrogante deberemos estudiar un tema más: el arrojo de excepciones por medio de la sentencia **throw**.

REGLAS DE LOS BLOQUES CATCH

Pueden poseer tantas líneas de código como se requiera (pero es una buena práctica mantenerlos breves). Sólo se acepta un **catch** sin especificación de clase de excepción. Cada **catch** puede tratar sólo un tipo de excepción. Las clases más específicas de excepción deben ir primero.

GENERAR EXCEPCIONES

Nuestros objetos también podrán generar excepciones. Si verificamos en el código alguna de nuestras funciones, podremos informar al método invocante que, por ejemplo, un parámetro no es válido. Para esto, se utiliza la sentencia **throw**.
Veamos un ejemplo. Supongamos que creamos un método que opera con un parámetro entero que debe ser impar. Si el método verifica que el número recibido no es impar, podría arrojar una excepción al método invocante que informe esto.

```
private void Calcula(int num)
{
    // Verifico que el número sea impar
    if (num % 2 == 0)
        throw new FormatException("El número debe ser impar");
    // Si sigo aquí es porque el número ES impar
    // ...
}
```

En este caso, la sentencia **throw** irá acompañada por el objeto del tipo excepción que mejor se adapte al tipo de error que ha ocurrido. También podríamos crear nuestra propia subclase de alguna excepción existente si lo consideráramos conveniente (**SystemException**, **FormatException**, etc.).
Cuando se ejecuta el **throw**, se abandona el método y se produce la excepción.
También es habitual que cuando se produce una excepción, ésta sea tratada y luego transferida al método superior:

```
int num, res;
try
{
    num = Convert.ToInt32(txtNum.Text);
    res = 5 / num;
}
catch (FormatException fe)
{
    // Tratamiento de la excepción de formato

    // ...
    throw fe;
}
```


Ahora sí podemos volver al caso del **finally**. ¿Por qué no era conveniente cerrar el archivo después de todo el bloque **try..catch**? Porque si se transfiere la excepción hacia el método invocante, el código del **finally** es ejecutado antes de abandonar el método, mientras que el código posterior al bloque **try..catch** no.

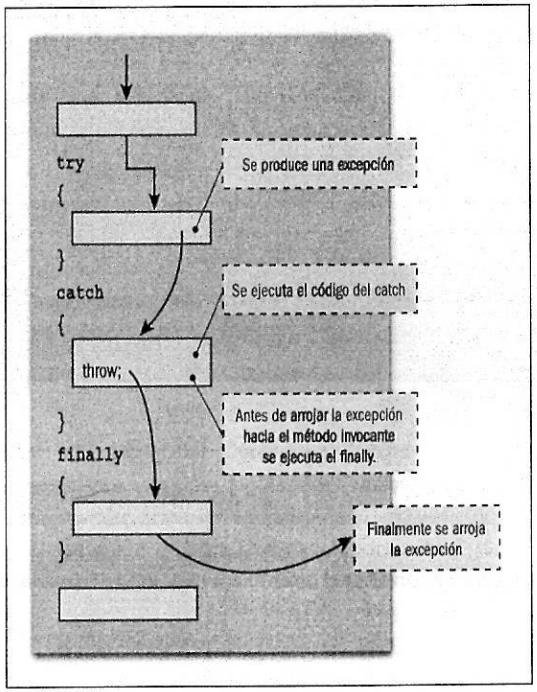


***Figura 9.** A pesar de existir un throw previo al finally, éste es ejecutado antes de abandonar el método.*

En el caso de que hubiéramos colocado el **cerrar archivo** inmediatamente después del bloque **try..catch**, éste habría quedado sin ejecutar.

EL CÓDIGO DENTRO DE LOS BLOQUES TRY Y CATCH

En ambos bloques, la cantidad de líneas de código que pueden contener no se encuentra limitada. Sin embargo, es una buena práctica mantenerlos breves.

CREAR NUESTRAS PROPIAS CLASES DE EXCEPCIÓN

Para crear nuestras propias clases de excepciones, simplemente debemos heredar de alguna excepción del tipo **SystemException** o descendiente de ésta.

Volviendo al ejemplo del número impar, podríamos pensar que **FormatException** tal vez no fuera el tipo de excepción más adecuado para describir lo que ha ocurrido. Por lo tanto, podríamos crear la siguiente clase:

```
public class NumImparException : FormatException
{
    public NumImparException() : base("El número no es impar.")
    {
    }
    public NumImparException(string strErr) : base(strErr)
    {
    }
}
```

En este caso se permite construir el objeto haciendo uso del mensaje predeterminado o personalizando el mensaje que será enviado.

Para arrojar la excepción ahora podríamos escribir:

```
private void Calcular(int num)
{
    if (num % 2 == 0)
        throw new NumImparException();
    // ...
}
```

Y ésta podría ser atrapada escribiendo:

```
try
{
    int num = Convert.ToInt32(txtNum.Text);
```



```
    Calcular(num);
}
catch (NumImparException nie)
{
    MessageBox.Show(nie.Message);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

RESUMEN

El manejo de excepciones es un poderoso sistema para especificar cómo debe comportarse nuestro programa cuando se produce algún evento no previsto. En el capítulo siguiente analizaremos más aspectos avanzados del lenguaje, entre ellos, uno de los más interesantes y atractivos de todos: Reflection.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Por qué es mejor utilizar el sistema de excepciones que el clásico sistema de errores retornados como valor de una función?

2 ¿Existe alguna desventaja del sistema de excepciones respecto al clásico pasaje de código de error por retorno de función?

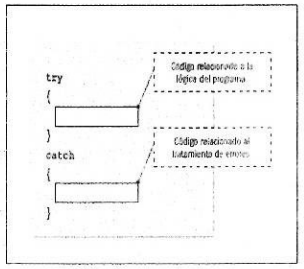
3 ¿Cuántos catchs podría poseer un bloque try..catch?

a. Cero o más.

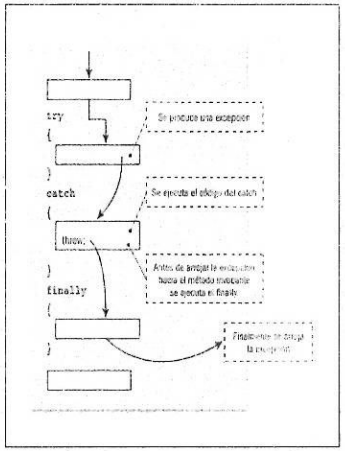
b. Uno solo.

c. Uno o más.

d. Más de uno.



4 ¿Qué ocurre si colocamos un catch de una excepción más general por encima de una excepción más específica?



5 ¿Cuántos catch sin parámetros son aceptados en un bloque try..catch? ¿Existe alguna regla que imponga su/s posicionamiento/s?

Capítulo 10

Características avanzadas del lenguaje

En este capítulo ingresaremos en los aspectos más avanzados con los que cuenta el lenguaje. Qué es el boxing/unboxing, qué son y para qué se utilizan los atributos, qué es Reflection y cómo acceder a librerías creadas en otros lenguajes.

BOXING/UNBOXING

Boxing es un proceso implícito que consiste en la inserción de un tipo de dato por valor en un objeto. Esto ocurre cuando se espera una referencia pero se provee un valor, entonces el framework automáticamente crea un objeto al vuelo, coloca dicho valor dentro de él y luego pasa la referencia a quien la esperaba.

```
int num = 10;
object obj = num;
```

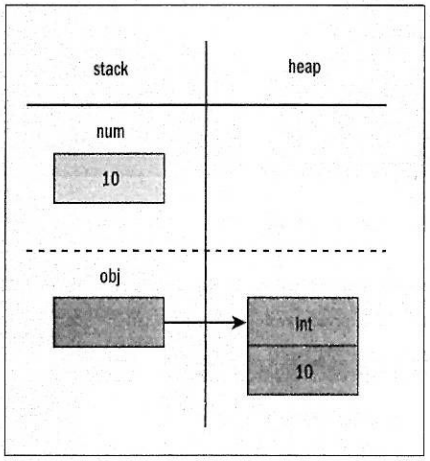


***Figura 1.** Boxing.*

El proceso contrario al **Boxing** es denominado **Unboxing**, y en contraposición al primero, éste **debe** ser explícito.

Para esto deberemos estar seguros de que dentro del objeto existe un tipo de dato por valor encerrado. Luego podremos escribir:

PRECAUCIONES CON BOXING/UNBOXING

Si estamos realizando una aplicación de alto desempeño, será importante conocer cuándo se realiza este tipo de operaciones implícitas, debido a que consumen algunos ciclos extra de CPU, especialmente, en zonas críticas como los bucles internos.


```
int num = (int) obj;
```

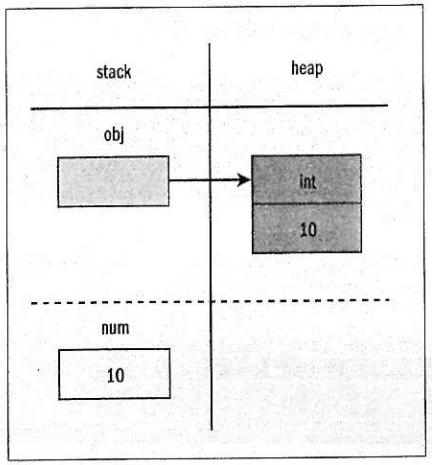


***Figura 2.** Unboxing.*

Si acaso intentáramos realizar un **unboxing** de un objeto que en realidad no posee un tipo de dato por valor encerrado dentro de él, entonces la plataforma arrojaría una excepción del tipo **InvalidCastException**.

ATRIBUTOS

Los metadatos son datos acerca de los datos. C# ofrece un sistema para insertar metadatos en nuestros assemblies mediante atributos.

Un atributo simplemente se define por medio de una clase descendiente de **System.Attribute**. Es bastante frecuente que el nombre que poseen estas clases descendientes

USO DE LOS ATRIBUTOS

Los atributos nos permiten fijar información de las estructuras que posee nuestro programa. Esto podría ser útil en muchos casos y, en particular, cuando nuestros programas sean inspeccionados mediante **Reflection** (tema que veremos un poco más adelante en este mismo capítulo).

termine en **Attribute** (como por ejemplo: **CualquierNombreAttribute**), aunque no se trata de una regla obligatoria.
Luego, a nuestra clase de tipo atributo podremos agregarle propiedades, pero por el momento analicemos un ejemplo bien sencillo:

```
class CualquierNombreAttribute : Attribute
{
}
```

De esta manera, hemos creado un atributo llamado **CualquierNombreAttribute**. A continuación veremos la manera de utilizarlo:

```
[CualquierNombreAttribute] class Foo
{
}
```

O simplemente:

```
[CualquierNombre] class Foo
{
}
```

Estos dos últimos listados de código son equivalentes, ya que el compilador permite simplificar el nombre del atributo quitando el **Attribute** detrás de él.

También podríamos agregar atributos a métodos, enumeradores, propiedades, eventos, etc. No obstante ello, un atributo específico podría restringir su rango de aplicación por medio de un parámetro de tipo **AttributeTargets** del atributo **AttributeUsage** a la definición de nuestra clase atributo.

MÁS ACERCA DE LOS ATRIBUTOS

En la documentación oficial que acompaña a Visual Studio .NET podremos encontrar más información sobre el uso de los atributos. También podremos acceder a dicha información online desde **http://msdn.microsoft.com/library/en-us/csref/html/vclrfintroductiontoattributes.asp**.


```
[AttributeUsage(AttributeTargets.Class)]
public class CualquierNombreAttribute: Attribute
{
}
```

Ahora, nuestro atributo **CualquierNombre** sólo es aplicable a clases.

También es posible combinar los **AttributeTargets** para hacer el atributo aplicable a más de un elemento:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class CualquierNombreAttribute: Attribute
{
}
```

Nuestro atributo **CualquierNombre** ahora es aplicable a clases y métodos.

VALOR	PUEDE PRECEDER A
All	Definiciones de cualquier elemento.
Assembly	Definiciones de assemblies.
Class	Definiciones de clases.
Constructor	Definiciones de métodos constructores.
Delegate	Definiciones de delegados.
Enum	Definiciones de enumeradores.
Event	Definiciones de eventos.
Field	Definiciones de campos.
Interface	Definiciones de interfaces.
Method	Definiciones de métodos.
Module	Definiciones de módulos.
Parameter	Definiciones de parámetros.
Property	Definiciones de propiedades.
ReturnValue	Definiciones de valores de retorno.
Struct	Definiciones de estructuras.

***Tabla 1.** Posibles valores de AttributeTargets.*

En las circunstancias donde estamos imposibilitados de determinar a qué elemento se debe aplicar el atributo, es posible indicarlo de modo explícito. Para ello debemos anteponer el nombre del elemento al nombre del atributo, como podemos observar en el siguiente diagrama de sintaxis:

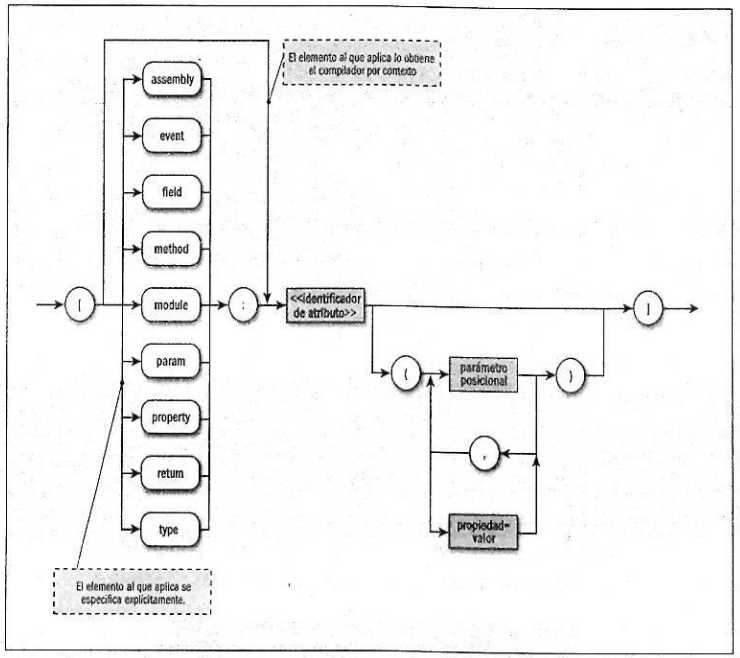


Figura 3. *Es posible especificar el elemento que aplica de modo explícito o esperar a que el compilador lo obtenga por contexto (aunque en ciertas ocasiones no puede hacerlo por existir ambigüedades).*

A continuación, agregaremos a nuestro atributo una propiedad capaz de ser fijada por medio del constructor:

```
[AttributeUsage(AttributeTargets.Class)]
public class CualquierNombreAttribute: Attribute
{
     private string valor;
     public CualquierNombreAttribute(string v) { valor = v; }
}
```

Y podremos utilizarlo del siguiente modo:

```
[CualquierNombre("prueba")] class Foo
{
}
```


También podríamos definir una propiedad pública, al igual que en cualquier clase:

```
[AttributeUsage(AttributeTargets.Class)]
public class CualquierNombreAttribute: Attribute
{
     private string valor1;
     private string valor2;
     public CualquierNombreAttribute(string v) { valor1 = v; }
     public string OtroValor
     {
          set { valor2 = value; }
          get { return valor2;  }
     }
}
```

Y luego modificar su valor del siguiente modo:

```
[CualquierNombre("prueba", OtroValor="otra prueba")] public class Foo
{
}
```

Atributos predefinidos

El **framework .NET** ofrece una serie de atributos predefinidos que pueden ser muy útiles en la creación de nuestros programas.

NOMBRE DE ATRIBUTO	APLICABLE A	DESCRIPCIÓN
Conditional	Método	En función de la evaluación de lo expresado en el atributo, se ejecuta o no el método relacionado.
DllImport	Método	Indica que el método se encuentra definido dentro de una librería de enlace dinámico con código no gestionado por la plataforma .NET.
Obsolete	Método	Permite marcar un método como obsoleto.

Tabla 2. *Algunos atributos predefinidos generales.*

Ejemplo de atributo condicional:

```
public class Foo
{
```

```
    [Conditional("DEBUG")]
    public static void EjecutaEnDebug()
    {
        MessageBox.Show("Ejecución de método EjecutaEnDebug");
    }
}
```

El atributo **Conditional** verificará si la constante de compilación condicional expresada se encuentra definida, y sólo de ser así se ejecutará el método relacionado.

La constante **DEBUG** es definida por el entorno sólo cuando la compilación se realiza en modo depuración. Si ahora escribimos:

```
Foo.EjecutaEnDebug();
```

Dicho método sólo se ejecutará realmente si el proyecto fue construido en modo **Debug**. Si, en cambio, fue construido en modo **Release**, dicha línea no formará parte del programa final.

Consultar los atributos en tiempo de ejecución

Hasta el momento nos hemos dedicado a explicar cómo crear nuevos atributos y cómo utilizarlos, aunque todavía no se ha evidenciado su verdadero poder o real utilidad. La información de los atributos que especifiquemos dentro de un assembly podrá ser consultada por un programa (disponible en el mismo assembly o en otro), el cual podrá modificar su flujo de ejecución en función de la información que encuentre dentro de los atributos.

De este modo, y por dar un ejemplo, podríamos **marcar** qué métodos deberían ser invocados bajo determinadas condiciones en cierto componente, sin que el assembly que utilice el componente conozca de antemano los nombres de los métodos por invocar ni su cantidad.

También podríamos ofrecer información transaccional de los objetos de un componente o información de utilidad al usuario, que por medio del uso de un sistema, pretenda utilizarlo dinámicamente.

Para poder acceder a esta información deberemos utilizar lo que en la plataforma .NET se conoce como **Reflection**.

REFLECTION

Reflection es un sistema que permite analizar el contenido de un assembly en tiempo de ejecución. Este sistema básicamente se utiliza para:

- Analizar la información de atributos.
- Descubrir nuevos tipos de datos e interactuar con ellos. De este modo podemos utilizar desde nuestro programa objetos desconocidos, listando sus propiedades, métodos, eventos, etc.
- Crear nuevos tipos en tiempo de ejecución.

Para consultar por la información de atributos que posee un assembly, deberemos:

1. Obtener una referencia al assembly que deseamos inspeccionar.
2. Listar los módulos que posee el assembly.
3. Listar los tipos (clases, interfaces, etc.) que posee cada módulo.
4. Listar los atributos que posee cada tipo (con la posibilidad de solicitar todos o el atributo del tipo específico que estamos buscando).

```
// Obtengo una referencia al assembly en ejecución
Assembly asse = Assembly.GetExecutingAssembly();
// Recorro todos los módulos del assembly
foreach (Module modulo in asse.GetModules())
{
    // Recorro todos los tipos del assembly
    foreach (Type tipo in modulo.GetTypes())
    {
        // Muestro en un cuadro de texto, el nombre del tipo actual
        txtLog.Text = txtLog.Text +
                      String.Format("Tipo: {0}\r\n", tipo.FullName);
        // Obtengo todos los atributos del tipo creado previamente
        //por nosotros
        CualquierNombreAttribute[] atts = (CualquierNombreAttribute[])
            tipo.GetCustomAttributes(
            typeof(CualquierNombreAttribute),
            false);
        // Recorro todos estos atributos
        foreach(CualquierNombreAttribute cna in atts)
            // Muestro en el cuadro de texto la
            // propiedad "OtroValor" del atributo
```

```
                txtLog.Text = txtLog.Text +
                        String.Format(" + Valor de atributo: {0}\r\n",
                                cna.OtroValor);
        }
}
```

Ejecutar métodos de tipos desconocidos

Mediante **Reflection** podremos conocer tipos nuevos en componentes cargados dinámicamente. También podremos invocar a voluntad métodos de estos tipos desconocidos.

Para ejemplificar esto, observemos una pequeña aplicación que permite cargar un assembly seleccionado por archivo, luego de lo cual inspecciona y ofrece sus módulos, tipos y métodos con el fin de que el usuario pueda seleccionar, finalmente, qué método invocar. Cargando el assembly:

```
dlgAbrirArchivo.Title = "Abrir assembly";
dlgAbrirArchivo.Filter = "Assembly (*.dll)|*.dll";
if (dlgAbrirArchivo.ShowDialog() == DialogResult.OK)
{
    txtAssembly.Text = dlgAbrirArchivo.FileName;
    BuscarModulos(dlgAbrirArchivo.FileName);
}
```

Aquí obtenemos el nombre del assembly seleccionado a través de un diálogo de apertura de archivo, e invocamos el método **BuscarModulos**:

```
private void BuscarModulos(string nombreAssembly)
{
```

III EL ESPACIO DE NOMBRES SYSTEM.REFLECTION

Para poder utilizar las clases **Assembly** y **Module** será conveniente agregar la línea **using System.Reflection** junto a las otras directivas **using** al comienzo del archivo de código fuente.


```
    Assembly asse = Assembly.LoadFile(nombreAssembly);
    if (asse != null)
    {
        // Se ha cargado el assembly correctamente
        // Cargo los módulos
        modulos = asse.GetModules();
        lsbModulos.Items.Clear();
        foreach(Module m in modulos)
            lsbModulos.Items.Add(m.Name);
    }
}
```

Utilizamos el método estático **LoadFile** de la clase **Assembly** para cargar un assembly y obtener una referencia a él.

Luego, por medio del método **GetModules** del objeto assembly cargado, obtendremos todos sus módulos, que colocaremos en una lista para que el usuario pueda seleccionar de qué módulo listar los tipos.
Para listar los tipos de un módulo escribiremos:

```
private void lsbModulos_SelectedIndexChanged(object sender,
    System.EventArgs e)
{
    if (modulos.Length > 0)
    {
        tipos = modulos[lsbModulos.SelectedIndex].GetTypes();
        if (tipos.Length > 0)
        {
            lsbTipos.Items.Clear();
            foreach(Type t in tipos)
                lsbTipos.Items.Add(t.Name);
        }
    }
}
```

Por medio del método **GetTypes** del objeto módulo seleccionado en la lista, obtendremos todos los tipos que insertaremos dentro de otra lista.
Ahora, finalmente sólo nos resta listar los métodos del tipo seleccionado y colocarlos en otra lista. Para esto deberemos escribir en el código:

```
private void lsbTipos_SelectedIndexChanged(object sender,
    System.EventArgs e)
{
     if (tipos.Length > 0)
     {
          metodos = tipos[lsbTipos.SelectedIndex].GetMethods();
          lsbMetodos.Items.Clear();
          foreach(MethodInfo minfo in metodos)
               lsbMetodos.Items.Add(minfo.Name);
     }
}
```

El usuario, entonces, se encontrará posibilitado de elegir el método que desee y presionar el botón **Invocar...**:

```
private void cmdInvocar_Click(object sender, System.EventArgs e)
{
     MethodInfo ms = tipos[lsbTipos.SelectedIndex]
        .GetMethod(nombreMetodoInvocar);
     if (ms != null)
     {
          try
          {
               ms.Invoke(null, null);
          }
          catch(Exception se)
          {
               MessageBox.Show(se.Message);
          }
     }
     else
          MessageBox.Show("No es posible encontrar el método
             especificado");
}
```

La única restricción que presenta nuestro ejemplo es que solamente está posibilitado para invocar métodos estáticos que no posean parámetros (situación que podría modificarse de manera bastante sencilla). De otro modo se producirá una excepción que luego se mostrará en un cuadro de texto.

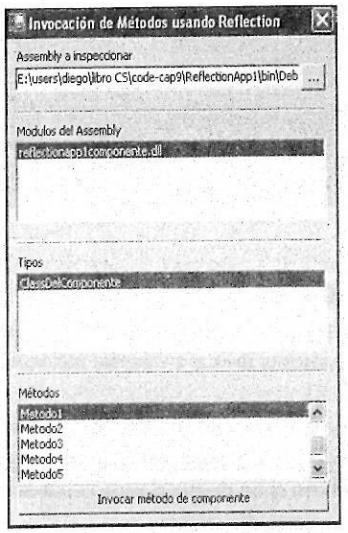


Figura 4. *Nuestro pequeño ejemplo en acción.*

ACCESO A OTRAS LIBRERÍAS

Es posible acceder desde un programa .NET a librerías de enlace dinámico creadas en otros lenguajes como C. Para esto se debe crear una clase que relacione por medio de métodos estáticos a las funciones globales que posee la librería.

Supongamos que dentro de una librería de enlace dinámico, que llamaremos **libcpp.dll**, creada en C, disponemos de las siguientes funciones:

```
extern "C" LIBCPP_API void fn1(void);
extern "C" LIBCPP_API int fn2(void);
extern "C" LIBCPP_API void fn3(char * pszText);
```

La primera función no recibe parámetro y nada retorna. Cuando es invocada muestra un cuadro de texto con información arbitraria.
La segunda función no recibe parámetros pero devuelve un número entero. Cuando es invocada devuelve una constante arbitraria.
La tercera función recibe un **char***, que deberá ser un array de caracteres de un byte terminado en **\0** (convención de strings en lenguaje C). Cuando es invocada, muestra un cuadro de texto con la información pasada como parámetro.

Ahora observemos, en el siguiente código, cómo debe ser nuestra clase en **C#** para permitirnos el acceso a la librería.

```
using System;
using System.Runtime.InteropServices;
public class LibCpp
{
    [DllImport("libcpp.dll", CharSet=CharSet.Auto, SetLastError=true)]
    public static extern void fn1();
    [DllImport("libcpp.dll", CharSet=CharSet.Auto, SetLastError=true)]
    public static extern int fn2();
    [DllImport("libcpp.dll", CharSet=CharSet.Auto, SetLastError=true)]
    public static extern void fn3(byte[] pszText);
}
```

La clase en cuestión posee un método estático por cada función mapeada. Cada uno de estos métodos posee un atributo preestablecido por el framework que nos permite relacionar la función con la librería.

Cada método debe poseer el mismo prototipo que la función en C que deseamos invocar. Es decir que debemos mapear los tipos de datos a los que mejor se adapten en lenguaje **C#**. Si uno de dichos parámetros es una estructura, podremos declarar una análoga en **C#**, pero siempre deberemos tener en cuenta que en memoria las variables internas de la estructura deben mapearse exactamente igual (hay que tener mucho cuidado con la alineación de los miembros de la estructura).
Hecho esto, las funciones podrán ser utilizadas como si fuesen nativas.

Invocar la primera función:

```
LibCpp.fn1();
```

Invocar la segunda función:

```
// Invoco la función
int num = LibCpp.fn2();
txtRetorno.Text = num.ToString();
```

Invocar la tercera función:


```
// Paso de string a un array de caracteres de 1 byte
byte[] txt = new byte[txtParam.Text.Length + 1];
for (int i=0; i<txtParam.Text.Length; i++)
    txt[i] = (byte) txtParam.Text[i];
txt[txtParam.Text.Length] = (byte) '\0';
// Invoco la función
LibCpp.fn3(txt);
```

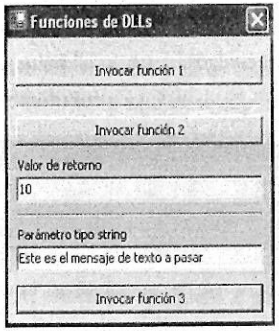


***Figura 5.** Nuestro ejemplo en acción.*

PUNTEROS

Un puntero es una variable que posee como valor una dirección de memoria asignada y desasignada explícitamente.
Este tipo de recursos es muy utilizado en los lenguajes de programación C y C++, debido al inmenso poder que otorgan, pero poseen una importante desventaja: es fácil cometer errores en su manipulación.

Los tipos de errores más comunes son:

- **Errores de lectura o escritura en zonas de memoria no permitidas**: si intentamos leer o escribir memoria que no nos pertenece, **Windows**, en forma preventiva, cerrará inmediatamente nuestra aplicación. A este error se lo conoce como **"error de protección general"**.
- **Errores en la desasignación de memoria**: si no liberamos la memoria de un modo correcto, entonces un sector de ésta quedará tomado por nuestra aplicación pero no será utilizado (*memory leak*). Si este error se comete repetidas veces en una aplicación, tarde o temprano se consumirá todo el recurso de memoria de la plataforma.

Por otro lado, estos tipos de errores suelen ser muy difíciles de extraer. Es por ello que **C#**, al igual que **Java**, incorporó un gestor automático de memoria para evitarle al programador esta tediosa tarea.

Pero **C#** ofrece la posibilidad de utilizar punteros, si lo solicitamos explícitamente. Para esto debemos hacerlo en un contexto no seguro (encerrado de un bloque **unsafe**); además, deberemos modificar una opción de nuestro proyecto para permitir código no seguro, y también debemos tener en cuenta que ciertas plataformas podrían no permitir la ejecución de aplicaciones con código no seguro en su interior.

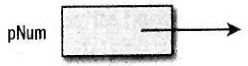
La operación con punteros es algo diferente del manejo de variables convencionales. Observemos la tabla que se incluye a continuación.

OPERADOR	DESCRIPCIÓN
&	El operador & permite obtener la dirección de memoria donde se encuentra alojada la variable relacionada.
*	El operador * permite acceder al valor apuntado por un puntero.
->	El operador -> permite acceder a los miembros de una clase o estructura apuntada por un puntero.

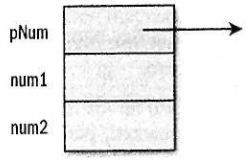
***Tabla 3.** Manejo de operadores en los punteros.*

Veamos un pequeño ejemplo del uso de punteros en **C#**:

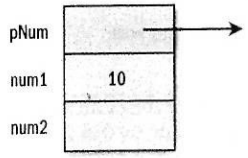
int * pNum;



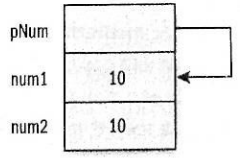
int num1, num2;



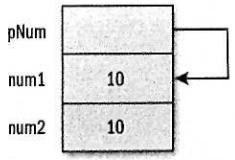
num1 = 10;



pNum = &num1;



num2 = *pNum;



Otra de las precauciones que debemos tener en cuenta a la hora de trabajar con punteros se encuentra relacionada con la gestión automática de memoria que poseen los elementos a los cuales apuntamos con los punteros.
Si el framework .NET entiende que una variable queda fuera de alcance, podría querer reutilizar la memoria que ésta ocupa, y de este modo, invalidar tal vez algún puntero que la estaba utilizando.
Para evitar esto se utiliza la sentencia **fixed** de la siguiente manera:

```
int[] nums = new int[10];
fixed (int * pNum = nums)
{
    //
}
```

De este modo prevenimos al framework .NET de reutilizar memoria con la cual aún estamos trabajando.
Los punteros no son muy populares en C#. A decir verdad, raramente veremos que se utilicen más allá de la necesidad inevitable por interactuar con componentes no gestionados que esperen punteros.

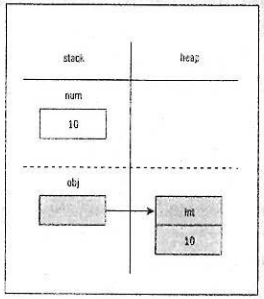
RESUMEN

En este capítulo hemos analizado algunas de las características más avanzadas que posee el lenguaje. Las posibilidades que nos brinda el mecanismo de Reflection son enormes. Muy fácilmente podríamos incorporar a nuestro programa un sistema de scripting por medio del cual el usuario pudiera extender las funcionalidades de la aplicación (por ejemplo, como si fueran macros de Microsoft Word), y todo utilizando C#, lo que permitiría, sin esfuerzo alguno, que se instanciaran clases de nuestro sistema. Por otro lado, la posibilidad de utilizar librerías creadas en otros lenguajes es una característica que muchos programadores C/C++ aplaudirán, ya que podrán reutilizar años de trabajo; incluso podrían decidir realizar ciertas tareas críticas en C mientras que todo el front-end de la aplicación se ejecutara en C#.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Cuáles son las ventajas y las desventajas de Boxing/Unboxing?



2 Cuando creamos una aplicación de tipo Windows Application con Visual Studio .NET, ¿dónde puede apreciarse el uso de atributos?

3 ¿Es posible instanciar objetos desconocidos de un assembly cargado en tiempo de ejecución?

4 ¿Qué precauciones se deben tener en cuenta en el mapeo de tipos de datos hacia librerías externas?

5 ¿Por qué se desaconseja el uso de punteros C# y, sin embargo, se ofrece la posibilidad de utilizarlos?

Capítulo 11

Servicios web

Los servicios web son uno de los aspectos de la plataforma que más han crecido en popularidad en los últimos tiempos. Si bien no son una característica exclusiva de .NET, Microsoft ha impulsado mucho esta tecnología y ha hecho un gran trabajo en facilitar la tarea de su construcción y utilización desde su entorno de desarrollo. Veremos aquí de qué trata esta tecnología y cómo podremos utilizarla desde nuestro querido C#.

¿QUÉ SON LOS SERVICIOS WEB?

Los servicios web (**web services**) permiten crear aplicaciones distribuidas colocando objetos en distintos puntos de la Red que publiquen servicios, que luego otras aplicaciones puedan consumir de una manera muy sencilla y, prácticamente, transparente.

Estos servicios, además, permiten originar arquitecturas no dependientes de la plataforma, ya que se basan en protocolos abiertos. Una aplicación .NET podría estar consumiendo servicios creados en **Java**, o viceversa.

Un servicio posee asociada una dirección que identifica al recurso en la Red y provee una descripción de su interfaz, por medio de la cual los clientes pueden conocer los métodos que ofrece dicho servicio.

Crear una aplicación en .NET que consuma servicios es muy sencillo. Veamos, por ejemplo, cómo crear una aplicación **C#** que use el servicio de búsqueda de **Google**.

Servicio de búsqueda de Google

Google (**www.google.com**) ofrece un servicio web para que podamos realizar búsquedas (sin fines comerciales, claro) desde nuestros sitios o aplicaciones.

Para poder comenzar a trabajar con él, en primer lugar requeriremos una descripción formal del servicio. Esta descripción se especifica en un archivo **WSDL**.

Descargaremos el **API** (que contiene el archivo **WSDL** del servicio, documentación y aplicaciones de ejemplo) desde **www.google.com/apis**.

Hecho esto, podremos aprovechar la ocasión y obtener una cuenta Google, que nos será requerida para usar el servicio (es un modo de autenticación que le permite a **Google** controlar quién y cómo utiliza su servicio).

¿QUÉ ES WSDL?

WSDL (*Web Service Description Language*) es un formato estándar para definir servicios web. Se encuentra expresado en **XML**, y establece cómo acceder al servicio y qué operaciones se pueden realizar con él. Enlace relacionado: **www.w3.org/TR/wsdl**.

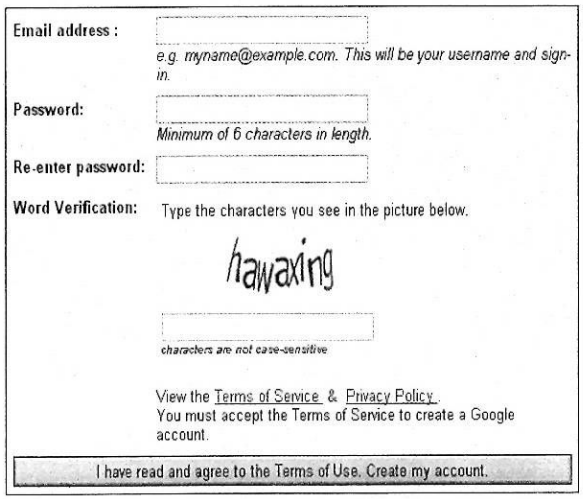


***Figura 1.** Los datos que solicita **Google** para crear una cuenta son realmente pocos.*

Nuestra cuenta nos permitirá realizar hasta mil consultas diarias, algo considerado más que suficiente para fines personales.

Creada la cuenta **Google**, nos enviarán por correo una licencia consistente en una clave alfanumérica de 32 caracteres. Esta clave es importante, ya que nos será solicitada cuando deseemos ejecutar una consulta utilizando el servicio web.

Ahora, veamos los pasos que deberemos realizar con **Visual Studio .NET** para poder interactuar con el servicio.

Crearemos una aplicación de tipo **Windows Application**, como lo venimos haciendo desde el **Capítulo 7**. Esta aplicación poseerá un solo formulario y, básicamente, un control de cuadro de texto (que servirá para especificar la consulta), un botón (que usaremos para ejecutar la consulta) y una lista de visualización (donde serán mostrados los resultados de la consulta).

Pero el paso más importante y clave que nos posibilitará utilizar el servicio web será agregar una referencia web a nuestro proyecto. Por medio de esta referencia, le indicaremos a **Visual Studio .NET** el servicio que deseamos utilizar mediante la especificación de un archivo **WSDL**.

Para agregar dicha referencia seleccionaremos la opción **Add Web Reference** desde el menú **Project** o desde el menú contextual sobre el panel **Solution Explorer**:

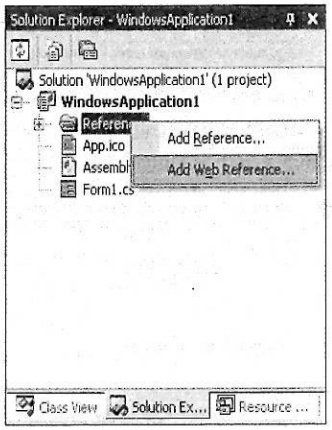


***Figura 2.** Agregando una referencia web.*

Una vez realizado esto, se abrirá una ventana que nos solicitará el ingreso de la ubicación de la descripción del servicio:

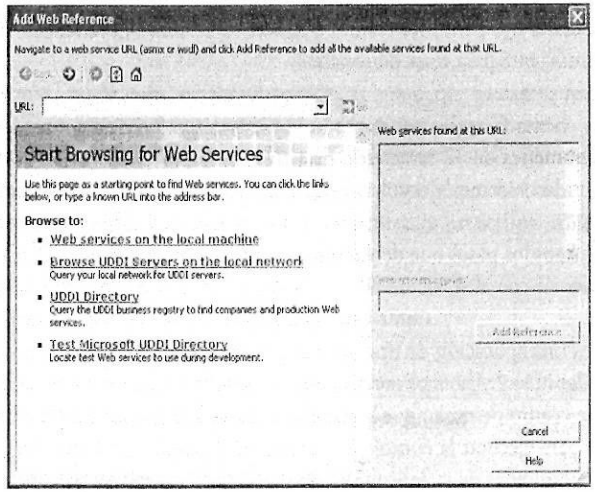


***Figura 3.** Debemos especificar la ruta de acceso al archivo WSDL descargado de **Google**.*

El archivo **WSDL** se encuentra ubicado dentro del paquete que hemos descargado de **Google**. Hay que descomprimir dicho paquete y luego especificar la ruta completa de acceso al archivo en cuestión, luego de lo cual **Visual Studio .NET** lo leerá y encontrará los métodos que ofrece el servicio:

- **doGetCachedPage**: devuelve una página dentro de la caché de **Google**.
- **doGoogleSearch**: establece una búsqueda por los criterios especificados.
- **doSpellingSuggestion**: realiza una sugerencia de escritura de los términos enviados.

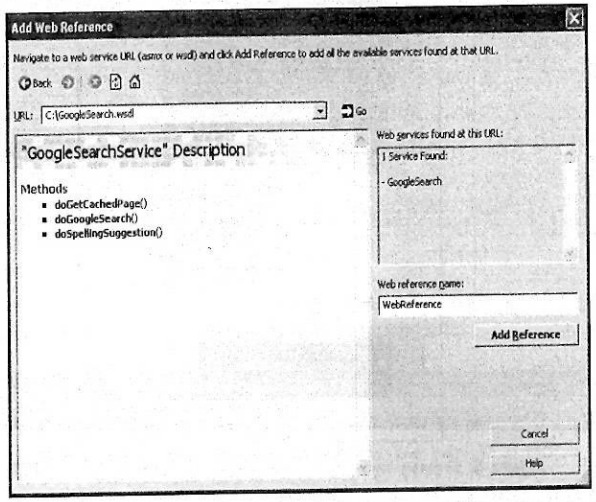


***Figura 4.** Los métodos encontrados en el servicio **GoogleSearchService**.*

Si en la ventana mostrada en la figura anterior presionamos el botón **Add Reference**, el entorno de desarrollo creará una clase **proxy** que nos permitirá interactuar con el servicio como si fuera un componente local, al tiempo que emplea el protocolo **SOAP** para dialogar con el servicio web.

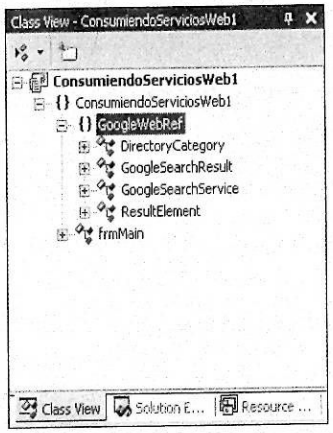


***Figura 5.** En el explorador de clases podemos apreciar la referencia web incorporada a nuestro proyecto.*

De este modo, nuestra aplicación interactúa con el servicio de modo transparente:

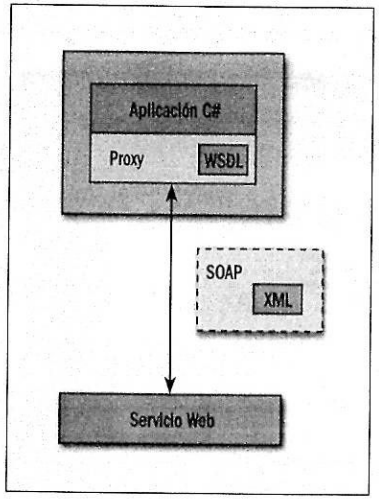


***Figura 6.** El proxy nos evita tener que trabajar con el protocolo SOAP.*

Utilizar llamados sincrónicos

Ahora llegó el momento de codificar. Principalmente, haremos uso del método **doGoogleSearch** de la clase **GoogleSearchService**.
El método mencionado cuenta con los siguientes parámetros:

NOMBRE	DESCRIPCIÓN
key	Clave de cuenta provista por Google como medio de autentificación.
q	Términos a buscar. Este texto puede utilizar ciertos operadores para ajustar los resultados de la búsqueda.
start	Índice (comenzando por cero) del primer resultado para devolver.
maxResults	Cantidad máxima de resultados devueltos (el máximo es 10).
filter	Permite filtrar resultados similares o que provengan del mismo sitio.
restrict	Permite restringir la búsqueda a un subconjunto especificado por un índice web de Google (como un país o una tecnología muy popular).
SafeSearch	Indica si se deben filtrar enlaces a sitios con contenido adulto.
lr	Restringe la búsqueda a un idioma.
ie	Parámetro obsoleto. Estaba relacionado con la codificación de caracteres utilizada para la entrada de datos.
oe	Parámetro obsoleto. Estaba relacionado con la codificación de caracteres utilizada para la salida de datos.

***Tabla 1.** Parámetros del método doGoogleSearch.*

```
GoogleSearchService gss = new GoogleSearchService();
GoogleSearchResult gsr = gss.doGoogleSearch(
```



```
    clave_otorgada_por_Google,      // clave
    txtSearch.Text,                 // texto a buscar
    0,                              // índice de comienzo
    10,                             // cant. máx. de resultados
    true,                           // filtro
    String.Empty,                   // índice web de google
    true,                           // búsqueda segura
    String.Empty,                   // restricción por idioma
    String.Empty,                   // Parámetro obsoleto
    String.Empty);                  // Parámetro obsoleto
```

Por otro lado, el resultado devuelto por la búsqueda se incluye en un objeto del tipo **GoogleSearchResult**, que posee las siguientes propiedades:

NOMBRE	DESCRIPCIÓN
documentFiltering	Valor booleano que indica si se practicó un filtrado de la información (esto lo habíamos especificado en la solicitud de búsqueda).
searchComments	Texto para mostrarle al usuario final que indica, por ejemplo, si se han quitado palabras especificadas en la búsqueda por ser muy comunes (como los artículos).
estimatedTotalResultsCount	Cantidad total estimada de resultados que arrojó la búsqueda (no la cantidad enviada hacia nosotros, que como máximo es 10, sino el total que emitió la búsqueda por la base de datos).
estimatedIsExact	Valor booleano que especifica si lo devuelto por estimatedTotalResultsCount es exacto.
resultElements	Array de elementos de tipo resultElements que contiene los resultados de la consulta.
searchQuery	Términos (parámetro q) con los cuales se especificó la consulta.
startIndex	Índice del primer resultado (comenzando por uno).
endIndex	Índice del último resultado.
searchTips	Texto para mostrarle al usuario final, con consejos acerca de cómo utilizar Google.
directoryCategories	Un array de elementos de tipo directoryCategory.
searchTime	En texto, el tiempo en segundos que tardó la búsqueda.

***Tabla 2.** Propiedades del objeto GoogleSearchResult.*

III ¿QUÉ ES SOAP?

SOAP (*Simple Object Access Protocol*) es un protocolo basado en **XML** que estandariza el modo de interacción con los servicios web. Enlace relacionado: **www.w3.org/TR/soap**.

Para poder otorgarle la información al usuario sobre las páginas encontradas, deberemos recorrer el array **resultElement**, que posee las siguientes propiedades:

NOMBRE	DESCRIPCIÓN
summary	Si el resultado se encuentra en el directorio ODP, aquí se ofrece una descripción.
URL	La dirección a la página relacionada.
snippet	Un pequeño resumen del texto encontrado en la página que se relaciona con la búsqueda.
title	Título de la página relacionada.
cachedSize	Tamaño de la página en caché.
relatedInformationPresent	Valor booleano que indica si el término de consulta "related:" es válido para la URL.
hostName	Para la segunda página encontrada relacionada con el mismo host, en esta propiedad se hallará el nombre de éste.
directoryCategory	Categoría de directorio ODP.
directoryTitle	Título del directorio ODP.

***Tabla 3.** Propiedades del array resultElement.*

La búsqueda se realiza de modo sincrónico, por lo que el control a nuestro programa retornará cuando ésta haya finalizado, y entonces podremos mostrar los resultados del modo que mejor nos parezca.

```
for (int i=0; i<gsr.resultElements.Length; i++)
{
    // Extraigo algunos tags HTML
    gsr.resultElements[i].title =
        gsr.resultElements[i].title.Replace("<b>", "");
    gsr.resultElements[i].title =
        gsr.resultElements[i].title.Replace("</b>", "");
    // Agrego el resultado al ListView
    ListViewItem li = new ListViewItem(gsr.resultElements[i].title);
    li.SubItems.Add(gsr.resultElements[i].URL);
    lsvResults.Items.Add(li);
}
```

Utilizar llamados asincrónicos

También podríamos haber realizado la búsqueda de modo asincrónico utilizando la versión de los métodos que comienzan con **Begin** y especificando el delegado correspondiente para alertarnos del arribo de los resultados.

Si realizamos un llamado asincrónico, nuestra aplicación no quedará en espera de los resultados, sino que podrá seguir operando normalmente. Lo importante es analizar si tiene sentido que la aplicación quede esperando los resultados o si es más beneficioso para el usuario poder seguir operándola.

Si exploramos la clase del servicio **GoogleSearchService**, notaremos que, además de los métodos **doGetCachedPage**, **doGoogleSearch** y **doSpellingSuggestion**, existen otros con los mismos nombres, pero con prefijo **Begin** y prefijo **End**; es decir:

BegindoGetCachedPage y **EnddoGetCachedPage**
BegindoGoogleSearch y **EnddoGoogleSearch**
BegindoSpellingSuggestion y **EnddoSpellingSuggestion**

Es importante saber que esta "**versión**" de métodos la ha creado **Visual Studio**, y no son otros métodos del servicio web que ofrezca **Google**.

Ahora, si en lugar de invocar **doGoogleSearch** invocamos **BegindoGoogleSearch**, el flujo de ejecución pasará a la línea siguiente casi inmediatamente después de invocar el método (sin que aún los resultados se encuentren disponibles).
¿Y cómo sabremos cuándo los resultados se encuentran disponibles? ¡Otra aplicación ideal para los delegados!

Los métodos **Begin...** reciben dos parámetros extra después de la lista de parámetros normales al método en cuestión. Éstos son:

PARÁMETRO	DESCRIPCIÓN
AsyncCallback callback	Delegado que será invocado cuando los datos arriben.
object state	Objeto que almacena información de estado. Será enviado al delegado cuando los datos arriben.

***Tabla 4.** Parámetros de los métodos Begin...*

Por lo tanto, nuestra llamada ahora podría modificarse a:

```
GoogleSearchService gss = new GoogleSearchService();
// Realizo un llamado asincrónico
gss.BegindoGoogleSearch(
    clave_otorgada_por_Google,          // clave
    txtSearch.Text,                     // texto a buscar
    0,                                  // índice de comienzo
    10,                                 // cant. máx. de resultados
    true,                               // filtro
    String.Empty,                       // índice web de google
    true                                // búsqueda segura
```

```
        String.Empty,                       // restricción por idioma
        String.Empty,                       // Parámetro obsoleto
        String.Empty,                       // Parámetro obsoleto
        new AsyncCallback(ConsultaTerminada), // Delegado
        gss);                               // estado
```

ConsultaTerminada será el método que indicamos que se invoque cuando los datos se encuentren disponibles:

```
public void ConsultaTerminada(IAsyncResult ar)
{
    GoogleSearchService gss = (GoogleSearchService) ar.AsyncState;
    // Recupero el objeto tipo GoogleSearchResult
    GoogleSearchResult gsr = gss.EnddoGoogleSearch(ar);
    // Mostramos los resultados del modo usual
    // ...
}
```

Como se puede apreciar en el listado anterior, primero recuperamos nuestro objeto **GoogleSearchService**, que se encontraba encapsulado en el resultado de la operación asincrónica. Luego obtenemos el resultado que antes devolvía el método sincrónico, utilizando para tal fin el **EnddoGoogleSearch**.
Finalmente, mostramos los resultados como mejor nos parezca.

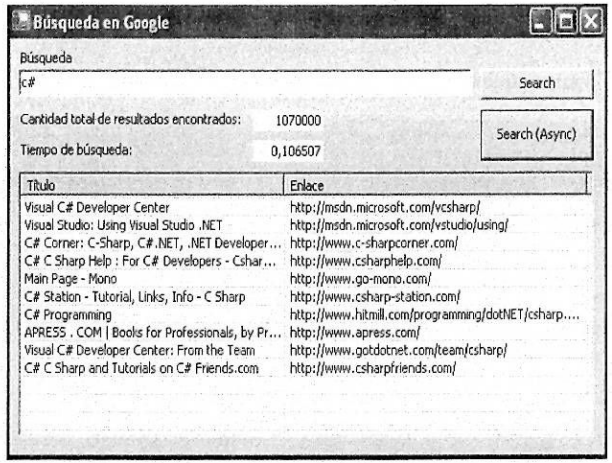


***Figura 7.** Nuestra simple aplicación de búsqueda en **Google**.*

CREAR UN SERVICIO WEB

Es interesante consumir servicios web, pero mucho más lo es crear nuestros propios servicios, lo que nos permitirá distribuir componentes de nuestra aplicación en una red. Veamos paso a paso qué procedimiento seguir para cumplir nuestro objetivo.

En primer lugar, debemos verificar que poseemos un servidor web **Microsoft IIS** (*Internet Information Server*) **5.0** o superior ejecutándose. El sistema operativo **Windows XP Professional** incluye el **IIS 5.1**, pero no es instalado por predefinición. Por lo tanto, hay que ir al panel de control, ingresar en la opción **Agregar o Quitar Programas**, desde allí elegir **Agregar o quitar componentes de Windows** y, finalmente, marcar la opción **Servicios de Internet Information Server** e instalarlo.
Con **IIS** instalado y en ejecución, crearemos un nuevo proyecto en **Visual Studio .NET**; esta vez, el tipo de proyecto seleccionado será **ASP.NET Web Service**:

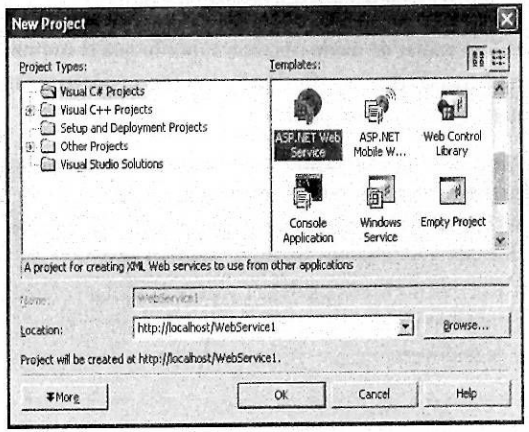


***Figura 8.** Creando un servicio web.*

Lo que hace en este caso el asistente es crear una carpeta dentro del sitio web que especificamos, y dentro de ella coloca los archivos que componen el proyecto.

IIS NO ESTÁ EJECUTANDO ASP.NET 1.1

Si cuando deseamos crear una aplicación del tipo **ASP.NET**, **Visual Studio** arroja un mensaje que indica que la versión de **IIS** que se encuentra en ejecución no posee instalada **ASP.NET 1.1**, probablemente deberemos ejecutar el comando **aspnet_regiis -r**.

Uno de estos archivos posee la definición de la clase principal de nuestro servicio web, que es una subclase de **System.Web.Services.WebService**:

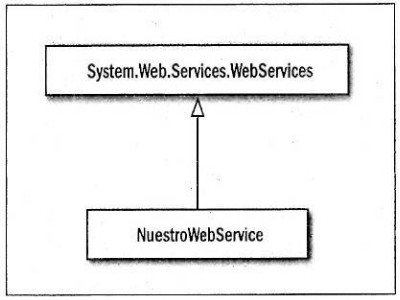


***Figura 9.** Nuestro servicio web debe ser descendiente de WebService.*

El código generado es bastante simple, y tiene un método web escrito, pero bajo comentarios, que podremos eliminar.

Nuestro ejemplo consistirá en crear cuatro métodos diferentes, cada uno de ellos con distintos tipos y cantidad de parámetros, simplemente para mostrar el mecanismo de funcionamiento de todo el sistema. Luego, estos métodos podrían trabajar como lo hace cualquier aplicación **C#** respecto al manejo de datos: accediendo a una base vía **ADO.NET**, o a archivos o a lo que nuestra aplicación requiera.

Todos los métodos que declaremos deberán poseer un atributo llamado **WebMethod** para que el compilador entienda que es un método para incluir en el servicio. Así, pues, nuestros métodos serán:

```
public class Service1 : System.Web.Services.WebService
{
    // Código generado por Visual Studio .NET
    // ...
```

EL TERCER PROTOCOLO: UDDI

Existe un tercer protocolo, muy utilizado en servicios web, pero no tanto como los otros dos (**SOAP**, **WSDL**). Se llama **UDDI** (*Universal Description, Discovery and Integration*), y permite localizar e interrogar servicios web para descubrir qué operaciones ofrecen y de este modo poder utilizarlas. Enlace relacionado: **www.uddi.org**.


```
    [WebMethod]
    public string Met1()
    {
        // Retorno un string arbitrario
        return "el lenguaje C#";
    }
    [WebMethod]
    public string Met2(string str)
    {
        // Retorno el string modificado
        return "Me has pasado el string: " + str;
    }
    [WebMethod]
    public int Met3(int num1, int num2)
    {
        // Retorno la suma de los parámetros
        return num1 + num2;
    }
    [WebMethod]
    public string[] Met4()
    {
        string[] strs = new string[4];
        strs[0] = "palabra1";
        strs[1] = "palabra2";
        strs[2] = "palabra3";
        strs[3] = "palabra4";
        // Retorno la suma de los parámetros
        return strs;
    }
}
```

¿DÓNDE ESTÁ EL ARCHIVO WSDL?

Es posible generar el contenido **WSDL** de la interfaz de nuestro servicio por medio del archivo **aspx** generado por el entorno, especificando **wsdl** como parámetro del siguiente modo: **http://localhost/WebService1/Service1.asmx?wsdl**. Suponiendo que nuestro proyecto se llamó **WebService1** y el servicio, **Service1**.

Ahora, compilaremos nuestra aplicación y la ejecutaremos. Hecho esto, se abrirá un navegador apuntando a un archivo del tipo **asmx** con el nombre de nuestro servicio.

Podremos, a través de esta página, llevar a cabo una prueba de los métodos de nuestro servicio web, que nos permitirá, incluso, especificar los parámetros que tenga el método que deseemos invocar.

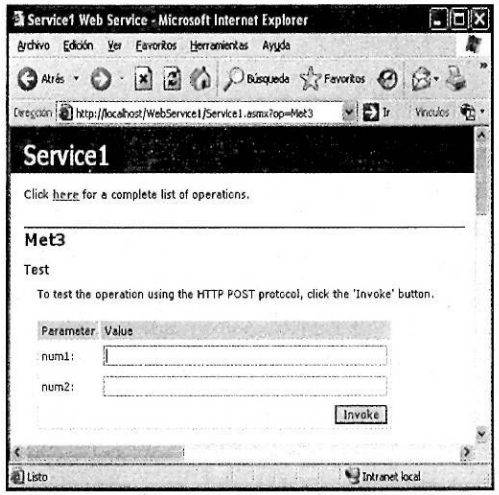


Figura 10. *Probando nuestro servicio web.*

Cuando invoquemos el método en cuestión, podremos inspeccionar el resultado **XML** que produce. En este caso, hemos invocado el método **Met3** con parámetros 5 y 4:

```
<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://www.dedalus-software.com.ar/">9</int>
```

Intentemos ahora desarrollar una pequeña aplicación que consume nuestro servicio, de la misma manera en que lo hemos hecho con el servicio de **Google**. Los pasos que deberemos seguir son muy sencillos:

1. Crearemos una aplicación tipo **Windows Application**.
2. Agregaremos una referencia web.
3. Especificaremos la ubicación del archivo **aspx** (en lugar del **wsdl** que habíamos utilizado en el ejemplo del servicio **Google**).
4. Instanciaremos la clase de nuestro servicio web y podremos invocar cualquiera de los métodos que posea.

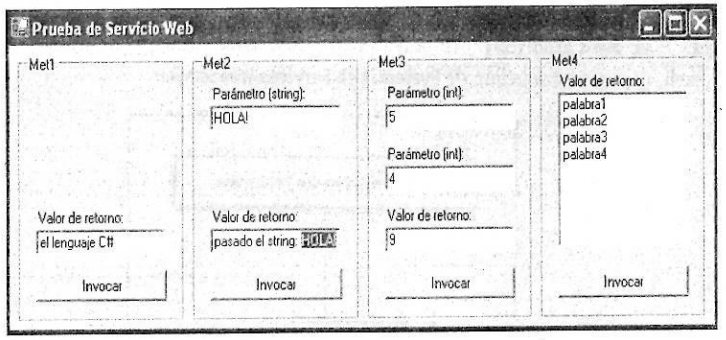


Figura 11. *Ejecutando los métodos de nuestro servicio web.*

Los archivos fuente completos de los ejemplos vistos en este capítulo pueden ser descargados desde el sitio **onweb.tectimes.com**.

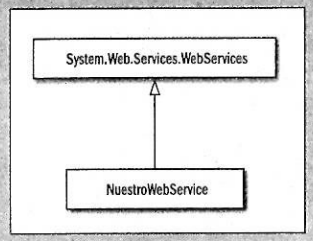
RESUMEN

Los servicios web son la tecnología de moda, y existen buenas razones para que lo sean. Muchas otras tecnologías persiguieron los mismos objetivos, y de hecho, fue posible conseguir logros similares mucho antes de que los servicios web vieran la luz. Sin embargo, nunca fue tan sencillo para el desarrollador construir y utilizar tecnología que permitiese distribuir, tan fácilmente, objetos por una red. Visual Studio .NET allana aún más el camino, convirtiendo en un placer la construcción de aplicaciones distribuidas.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Cuáles son los tres protocolos más utilizados en los servicios web?



2 ¿Qué ocurre si no agregamos el atributo WebMethod a los métodos de un servicio?

3 ¿Qué ventaja comporta el uso de métodos asincrónicos?

4 Un servicio web creado en ASP.NET ¿puede ser utilizado desde plataformas Unix?

5 ¿Cuál es la ventaja de crear aplicaciones distribuidas?

Capítulo 12

Sockets

Si deseamos que nuestra aplicación se comunique con otra por medio de una red de datos, varias soluciones son posibles. Una de ellas es la programación de sockets, que tienen la ventaja de ser fáciles de utilizar y de poder dotar nuestras aplicaciones de capacidades de red sin depender, por ejemplo, de servidores web.

SERVICIO DE ATENCIÓN AL LECTOR: lectores@mpediciones.com

¿QUÉ ES UN SOCKET?

Socket es una especificación que define una interfaz para la programación de aplicaciones de red. Se implementó inicialmente en sistemas operativos del tipo **BSD Unix** a principios de los años ochenta. Originalmente, la librería **Berkley Sockets** ofrecía una interfaz de programación en lenguaje **C**; hoy en día existen muchas implementaciones. Así, podemos encontrar la implementación sobre **MFC** llamada **CAsyncSockets**, la clase **Socket** en **.NET**, la clase **Socket** en **Java**, etc.

Dentro de un programa que haga uso de una librería de sockets, un socket es un extremo en la comunicación de dos programas a través de una red de comunicaciones.

Arquitectura de una aplicación de sockets

Haciendo uso de sockets, dos aplicaciones podrán comunicarse mediante una red. Para esto, es necesario que una de ellas oficie de servidor y comience a **escuchar** por un puerto específico, y la otra actúe de cliente y se **conecte** a la anterior.

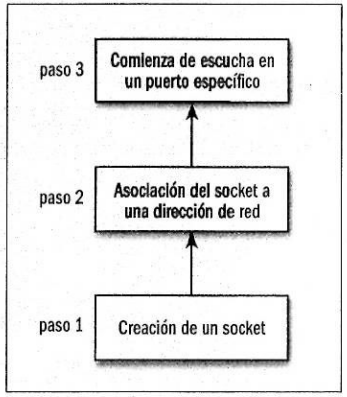


Figura 1. *Pasos en la creación de un servidor.*

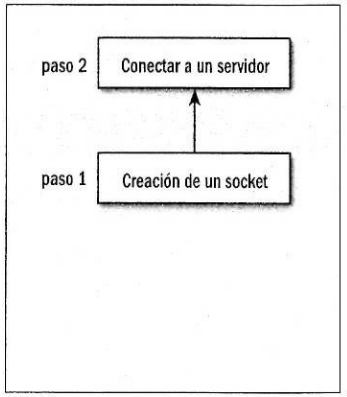


Figura 2. *Pasos en la creación de un cliente.*

Diálogo entre un cliente y un servidor

Cuando se realiza una conexión, existe un diálogo básico que se establece entre ambas partes y que determina si la conexión es aceptada o no.

Una vez que la conexión esté establecida, se comenzarán a intercambiar datos. El modo en que cada parte actuará en función de los datos entrantes estará determinado por el **protocolo de aplicación** que se defina.

Sincrónico vs. Asincrónico

La programación de sockets puede realizarse utilizando métodos sincrónicos o asincrónicos, algo similar a lo que ocurría con la invocación de métodos de un servicio web, y que vimos en el capítulo anterior. Los llamados sincrónicos son aquellos que esperarán a que la operación se complete para continuar con el flujo de ejecución del programa. Por ejemplo, si deseo conectarme a un servidor, invocaré el método **Connect**. Siendo el llamado sincrónico, en la línea posterior a la invocación del método el mismo habrá finalizado (de modo independiente al éxito de la operación).

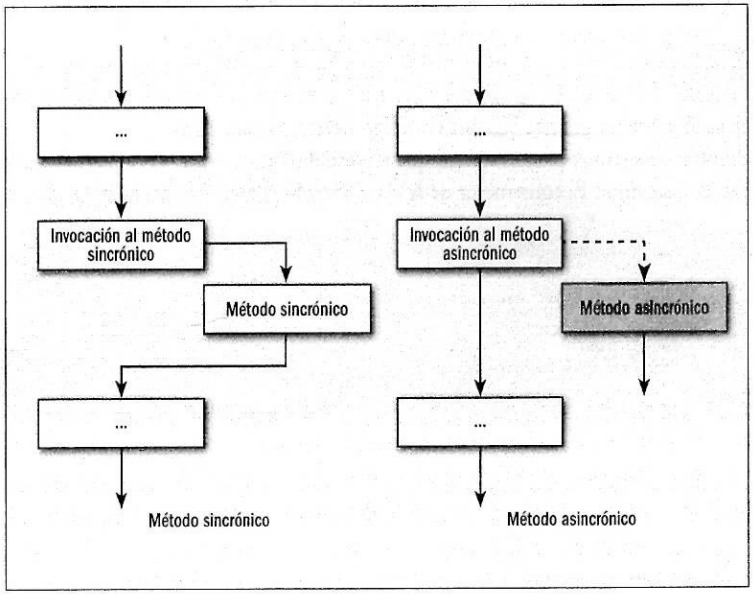


Figura 3. *Método sincrónico y asincrónico.*

Por el contrario, los métodos asincrónicos le indicarán a la librería **qué hacer**, pero el hilo de ejecución principal no se detendrá ante el correspondiente llamado. Citando un ejemplo similar al anterior, en este caso podríamos invocar **BeginConnect** y el programa pasaría a la línea siguiente sin que la operación necesariamente haya finalizado. ¿Cómo saber, entonces, si se realiza o no la conexión? Por medio de un llamado que hará la librería hacia el programa utilizando un delegado.

Está claro que si utilizamos llamados asincrónicos no deberemos hacer suposiciones sobre lo que sucede con el llamado en líneas posteriores a la invocación del método correspondiente. De modo asincrónico podremos conectarnos a servidores, aceptar conexiones de clientes remotos, enviar datos, recibir datos, etc.

La clase Socket

La clase **Socket** se encuentra dentro del espacio de nombres **System.Net.Sockets**; para hacer uso de ella, sin utilizar el nombre completamente calificado, agregaremos:

```
using System.Net.Sockets;
```

En este ejemplo, la aplicación oficiará de servidor o de cliente, de modo mutuamente excluyente. Esto significa que si disponemos de una sola computadora para probarla, tendremos que ejecutarla dos veces: una como servidor y otra como cliente.

El servidor

Habíamos mencionado los pasos necesarios para crear el servidor y dejarlo escuchando sobre un puerto. Veamos el código necesario para ello.
Primero deberemos crear el socket, que representará el extremo de la comunicación del lado servidor. El constructor de la clase **Socket** posee el siguiente prototipo:

```
public Socket(
    AddressFamily addressFamily,
    SocketType socketType,
    ProtocolType protocolType
);
```

El primer parámetro, **AddressFamily**, señala el esquema de direccionamiento que utilizará el socket. En nuestro caso, siempre indicaremos **AddressFamily.InterNetwork**. El segundo, **SocketType**, indica el tipo de socket que deseamos crear, que en nuestro caso será **SocketType.Stream**. Y el tercer parámetro marca el tipo de protocolo sobre el cual crearemos nuestro socket; aquí podríamos utilizar **ProtocolType.Tcp** o **ProtocolType.Udp**. Para nuestro ejemplo, nos decidiremos por **Tcp**:

```
Socket connSocket = new Socket(AddressFamily.InterNetwork,
    SocketType.Stream, ProtocolType.Tcp);
```

El segundo paso consistirá en asociar el socket a un determinado **endpoint**. Un **endpoint** representa el direccionamiento a un extremo de la comunicación, que en el protocolo que estamos utilizando está dado por una dirección **IP** y un puerto.
La dirección **IP** de nuestro servidor será alguna de las direcciones **IP** de la computadora que oficie como tal (por ejemplo, podremos utilizar la dirección **IP** especial **127.0.0.1** que hace referencia siempre al servidor local).


```
IPAddress hostIPAddress = IPAddress.Parse("127.0.0.1");
```

Respecto al puerto, fijaremos uno cualquiera; por ejemplo, el 8000:

```
IPEndPoint ipEndP = new IPEndPoint(hostIPAddress, 8000);
```

Con el objeto del tipo **IPEndPoint** creado realizaremos la asociación al socket utilizando el método **Bind** de la clase **Socket**:

```
connSocket.Bind(ipEndP);
```

Hecho esto, estamos listos para comenzar la escucha en el puerto especificado. Para ello utilizaremos el método **Listen**.

```
connSocket.Listen(1);
```

El método **Listen** recibe un parámetro que representa la cantidad de clientes que pueden estar conectados de manera simultánea al servidor.

Ahora podremos elegir qué métodos utilizar para aceptar la conexión entrante. Uno de ellos es **Accept**, que es sincrónico y que bloqueará nuestra aplicación hasta recibir una conexión entrante, y el otro es **BeginAccept**, que es asincrónico y con el cual deberemos indicarle qué método (de alguna de nuestras clases) se hará cargo de aceptar o no las peticiones entrantes. Optaremos por emplear el modo asincrónico, pues no deseamos que nuestra aplicación quede paralizada mientras espera clientes.

```
connSocket.BeginAccept(new AsyncCallback(OnAccept), connSocket);
```

El método **OnAccept** será del tipo **AsyncCallback**, y le pasaremos como parámetro el socket del servidor.

```
public void OnAccept(IAsyncResult ar)
{
    Socket connSocket = (Socket) ar.AsyncState;
m_remoteSocket = connSocket.EndAccept(ar);
```

```
        // Cambio el estado de la aplicación
        SetServerStatus(EServerStatus.svr_connected);
        m_remoteSocket.BeginReceive(m_receiveBuffer, 0,
            m_receiveBuffer.Length, SocketFlags.None, new
            AsyncCallback(OnReceive), m_remoteSocket);
    }
```

La línea de código más interesante del método **OnAccept** es:

```
m_remoteSocket = connSocket.EndAccept(ar);
```

Aquí, mediante **EndAccept** podremos acceder a otro socket, que representa el otro extremo de la conexión que aceptamos. Antes de finalizar el método, invocamos **BeginReceive** usando el flamante socket para comenzar a aceptar datos de la conexión.

El cliente

Para el papel de cliente de nuestra aplicación, deberemos crear un socket del mismo modo que lo hicimos para el servidor.

El próximo paso será conectarse al servidor, para lo cual haremos uso del método **Connect** de la clase **Socket**. Dicho método recibe como parámetro un **IPEndPoint**, como lo hacía el método **Bind**. En este caso, la dirección **IP** y el puerto serán arbitrarios, y en la aplicación resultarán elegidos mediante un **TextBox**.

```
IPAddress hostIPAddress = IPAddress.Parse(txtServerAddress.Text);
IPEndPoint ipEndP = new IPEndPoint(hostIPAddress,
    Convert.ToInt32(txtServerPort.Text));
try
{
    connSocket.Connect(ipEndP);
}

catch

{
    // No es posible conectarse
    // ...
}
```


Aquí utilizamos la versión sincrónica del método, ya que el tiempo estimado de conexión a un servidor es bajo, y mientras tanto la aplicación no debe realizar ninguna otra tarea. Esto significa que, invocado el método **Connect**, la aplicación quedará bloqueada durante unos instantes, mientras se está intentando la conexión. Si ésta fracasa, el método arrojará una excepción que deberemos manejar según corresponda (en la aplicación de ejemplo, simplemente anunciamos dicho fracaso mediante un **MessageBox**). Si la conexión fue satisfactoria, en la línea posterior al **Connect** asumiremos que existe una conexión establecida y podremos comenzar a intercambiar datos con el servidor.

Enviar y recibir datos

El envío y recepción de datos se realiza por medio de los métodos **Send** y **Receive**, respectivamente, ambos de la clase **Socket**. Los métodos mencionados son del tipo sincrónico, y poseen su contraparte asincrónica en **BeginSend** y **BeginReceive**. En nuestra aplicación de ejemplo haremos uso del envío de datos sincrónico (**Send**) y la recepción de datos asincrónica (**BeginReceive**). ¿Por qué esta disparidad? Debido a que el envío de datos comienza con una acción explícita por parte de nuestra aplicación, pero la recepción, no.
En concreto, los datos que intercambiará nuestra aplicación con otra instancia de sí misma serán mensajes de texto. Si analizamos el código que ejecuta esta aplicación cuando presionamos **Enviar**, veremos que lo que hace es armar un paquete de datos según un protocolo de aplicación básico que inventamos para la ocasión, lo convierte en un array de bytes y luego invoca el método **Send** para enviarlos. La conversión mencionada tiene que ver con el tipo de dato que dicho método espera como parámetro.
Nuestro paquete de información consta de una cabecera y una sección datos. La cabecera es fija y siempre son 2 bytes. El primero de ellos señala mediante un código el tipo de paquete de datos que llevará el cuerpo, y el segundo indica el tamaño (en bytes).

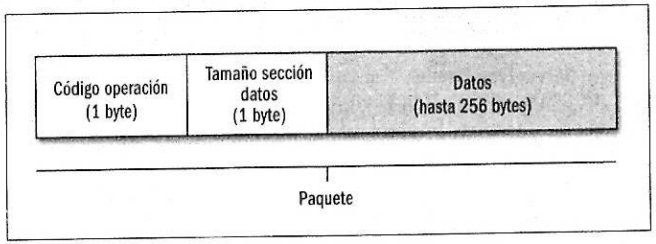


***Figura 4.** El paquete de datos inventado por nosotros.*

Para el ejemplo, aquí sólo haremos uso de un tipo de paquete (mensajes de texto), pero se podría extender su utilización para otro tipo de aplicaciones.
Respecto a la recepción de datos, ya dijimos que para manejar la operación emplearíamos un método asincrónico. Para ello, tendremos que invocar **BeginReceive** y pasar como parámetro qué método deberá manejar dicha recepción (similar a lo que hicimos con **BeginAccept**). Analicemos detalladamente el prototipo del método.

```
public IAsyncResult BeginReceive(
    byte[] buffer,
    int offset,
    int size,
    SocketFlags socketFlags,
    AsyncCallback callback,
    object state
);
```

Mediante el primer parámetro pasamos un buffer en el cual la librería colocará los datos recibidos. El segundo parámetro indica un **offset** (desplazamiento) en bytes a partir del cual escribirá en el buffer (usualmente colocamos 0). El tercero es el tamaño del buffer. El cuarto son flags relacionados con el socket. El quinto parámetro es el método que manejará la recepción de datos. Y, en último término, el sexto parámetro es utilizado con el fin de enviar información propietaria al método en cuestión.
Ya dentro de **OnReceive**, invocaremos el método **EndReceive**, que nos retornará la cantidad de bytes que recibió la librería (y que se encuentran en el buffer que pasamos como parámetro a **BeginReceive**).

```
public void OnReceive(IAsyncResult ar)
{
    Socket connSocket = (Socket) ar.AsyncState;
int read = connSocket.EndReceive(ar);
// ...
}
```

Recibidos los datos, nuestra aplicación los manejará de manera conveniente, y luego, si corresponde, antes de finalizar volverá a invocar **BeginReceive** para seguir permitiendo la recepción de datos.

Máquinas de estado

Con relación a los datos que recibe la aplicación y cómo los maneja, es frecuente que se implemente una máquina de estados. La máquina de la aplicación de ejemplo es bastante sencilla, y simplemente, se encarga de conmutar entre los distintos estados posibles del servidor y del cliente. Es importante que la aplicación implemente un mecanismo de este tipo, debido a que la recepción asincrónica de datos en ocasiones torna compleja la predicción de cómo se sucederán los hechos; por lo tanto, para que la aplicación se comporte consistentemente en todo momento, se debe verificar el estado en el que se encuentra antes de realizar una acción.

Los sockets nos permiten crear aplicaciones de red de manera muy sencilla y eficiente. Sin embargo, debemos destacar que no implementan ningún protocolo de aplicación en particular; ése será nuestro trabajo. Si deseamos comunicarnos mediante algún protocolo de aplicación particular (**HTTP**, **FTP**, etc.), quizá podamos utilizar otras clases de mayor nivel que ofrece la librería **BCL** del **framework .NET**.

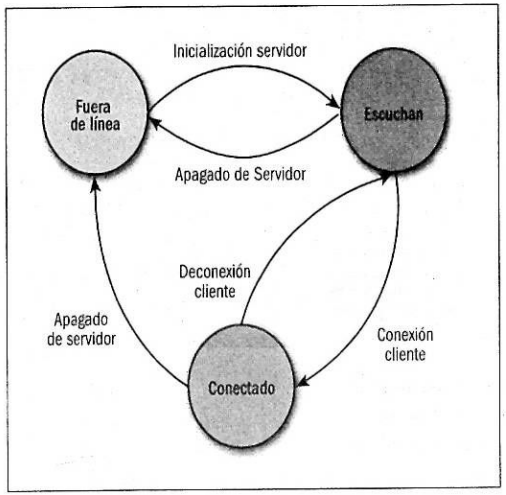


***Figura 5.** Estados posibles del servidor.*

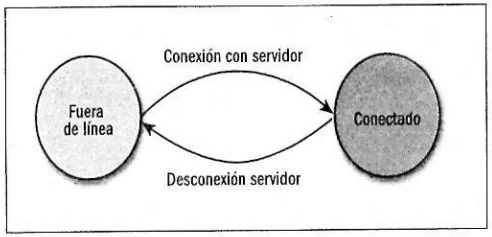


***Figura 6.** Estados posibles del cliente.*

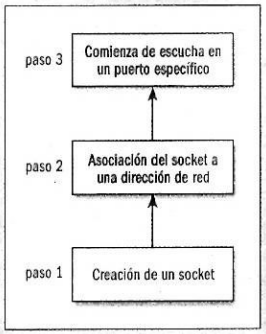
RESUMEN

Muchísimas aplicaciones populares hacen uso de sockets para comunicarse con otras mediante una red de datos como Internet. Éste es el caso de todas las aplicaciones P2P, como eMule, Kaaza, BitComet, BitTornado, etc. Programar con sockets es sencillo, y se trata de un recurso que siempre hemos de tener a mano como posible solución a requerimientos de conectividad que podrían tener nuestras futuras aplicaciones.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Es posible conectarse a un servidor web haciendo uso de la clase Socket e interactuar con él?



2 ¿Qué beneficios otorga la utilización de los métodos asincrónicos de la clase Socket? ¿Cuál es la contra?

3 ¿Requiere el servidor de una aplicación que utilice sockets la existencia de un servidor HTTP o SMTP?

4 ¿Es posible crear un servidor HTTP utilizando sockets? ¿Y un cliente HTTP como un navegador web?

5 ¿Sería posible consumir un servicio web por medio de una aplicación que utilice sockets?

Capítulo 13

Introducción a Managed DirectX

En este capítulo haremos una introducción a una de las APIs más importantes en el mundo de la programación de aplicaciones de alto rendimiento gráfico y juegos. Managed DirectX es una suma de clases para .NET que nos permitirán hacer uso de DirectX en su totalidad.

MANAGED DIRECTX

DirectX es un conjunto de componentes creado por **Microsoft** para ser utilizado en la creación de juegos y aplicaciones multimedia de alto desempeño.
Haciendo uso de esta herramienta podremos crear gráficos 2D y 3D; sonidos, música y efectos; aplicaciones de red, como son los juegos multiplayer, y manejar dispositivos de entrada como teclados, mouse, joysticks, volantes, game pads, etc.
Es posible hacer uso de él desde diversos lenguajes de programación como **C++**, **C#**, **Delphi**, **Visual Basic**, etc.

Más específicamente, desde la plataforma **.NET** deberemos utilizar **Managed DirectX** (**MDX**), que consiste en un conjunto de clases organizadas en espacio de nombres, que fue lanzada al público por primera vez el 20 de diciembre de 2002 y que pretende facilitar el desarrollo de juegos y aplicaciones haciendo uso de esta tecnología.

MDX es una gran API; en este capítulo nos enfocaremos concretamente en su componente más popular: **Direct3D**.

¿Qué necesitamos?

Si deseamos sólo ejecutar un programa **MDX**, necesitaremos:

- El framework .NET en su versión 1.1 o superior (**http://msdn.microsoft.com/netframework**). Podemos descargarlo desde el sitio de **Windows Update** o bien conseguir su versión redistribuible.
- Los runtimes de **DirectX 9.0c** o superior (**www.microsoft.com/windows/directx/default.asp**).

Si deseamos desarrollar aplicaciones haciendo uso de **Managed DirectX**, necesitaremos:

- El SDK del framework .NET 1.1 o superior (**http://msdn.microsoft.com/netframework**).
- El SDK del DirectX 9.0c SDK o superior (**www.microsoft.com/downloads/search.aspx?displaylang=en&categoryid=2**).

También es recomendable contar con algún entorno de desarrollo integrado como **Microsoft Visual Studio .NET**, que facilite la gestión del proyecto y su depuración.

Existe una base teórica fundamental que deberemos conocer, previamente a empezar a navegar por los mares de **Direct3D**. Comenzaremos con una breve reseña que nos será bastante útil como introducción a dicha teoría.

EL SISTEMA DE COORDENADAS 3D

Por medio de **Direct3D** lograremos crear una imagen en dos dimensiones a partir de un modelo tridimensional. Esto es así, naturalmente, porque nuestro monitor sólo puede generar una imagen chata de los datos procesados.

Además, **Direct3D** hará todo esto de modo muy eficiente utilizando características convenientes de la placa de video cuando éstas estén presentes, así como características del CPU que puedan mejorar el rendimiento de nuestra aplicación (por ejemplo, el conjunto de instrucciones **Intel MMX**, **SSE**, **SSE2** o el conjunto de instrucciones **AMD 3dNow!**).

Nuestros objetos 3D serán colocados en escena por medio de un sistema de coordenadas de tres dimensiones. **Direct3D** utiliza la convención de la mano izquierda que dicta que el eje **Z** crece positivamente hacia adentro de la pantalla; por lo tanto, un vértice que posea en su componente **Z** un valor mayor se encontrará más lejos que otro con valor menor.

Un objeto en **Direct3D** estará constituido exclusivamente por un conjunto de triángulos, los cuales a su vez estarán formados por un conjunto de vértices. Cada uno de estos vértices poseerá los tres componentes requeridos para ser colocados dentro de un sistema de coordenadas 3D, pero cabe mencionar que podrán tener otras propiedades en función de las características que deberá poseer el objeto del cual forme parte, y que, en su debido momento, analizaremos.

Cada vértice, antes de ser mostrado en pantalla, deberá pasar por una serie de transformaciones que dictarán si el vértice es finalmente mostrado y en qué posición de la pantalla estará. Estos cambios se darán dentro del denominado **pipeline**, cuyo diagrama podemos observar a continuación:

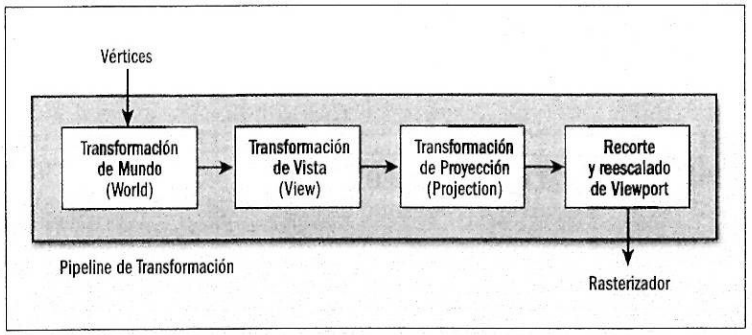


***Figura 1.** El pipeline fijo de transformación.*

El **pipeline** se alimenta de vértices, y cada uno será convertido a coordenadas homogéneas para luego poder ser multiplicado por tres matrices: la matriz de mundo, la matriz de vista y la matriz de proyección. Luego, la coordenada resultante es reescalada en función de puerto de visualización especificado (usualmente la pantalla completa o la ventana que estamos utilizando como objetivo para el dibujado), y en caso de quedar fuera del área de visualización, se elimina. Finalmente, el vértice es transferido al proceso de rasterización, donde será dibujado en pantalla.

La matriz de mundo
La primera matriz de mundo es utilizada con el propósito de colocar el objeto dentro de la escena 3D. Realiza una transformación que va desde el espacio del modelo al espacio del mundo en el cual trabajamos. La transformación de mundo puede incluir traslaciones, rotaciones y escalamientos.

La matriz de vista
La segunda matriz, llamada matriz de vista, reubica todos los objetos en función de un punto de visualización especificado.

La matriz de proyección
La tercera matriz, denominada matriz de proyección, especifica principalmente escala y perspectiva adoptada. Esta matriz determinará también cuánto vemos de la escena; es decir, el volumen de visualización.

ANTES DE EMPEZAR

Para hacer uso del API deberemos utilizar los espacios de nombre **Microsoft.DirectX** y **Microsoft.DirectX.Direct3D**, y si el compilador no logra encontrar estos recursos, es posible que falte agregar las referencias a él o exista un problema en la instalación del **SDK** de **DirectX 9.0c**.

DE DIRECTX A MANAGED DIRECTX

Los programadores que ya han utilizado DirectX descubrirán con grata sorpresa que Managed DirectX es casi idéntico, pero con una organización de clases mucho más conveniente, debido al uso de espacios de nombres. También aprovecha muchas de las capacidades de la plataforma (como los delegados) para facilitar la programación.

Establezcamos un objetivo: intentaremos crear una pequeña escena compuesta de una pirámide central, formada por cuatro triángulos, y por debajo, una plataforma compuesta por dos triángulos.

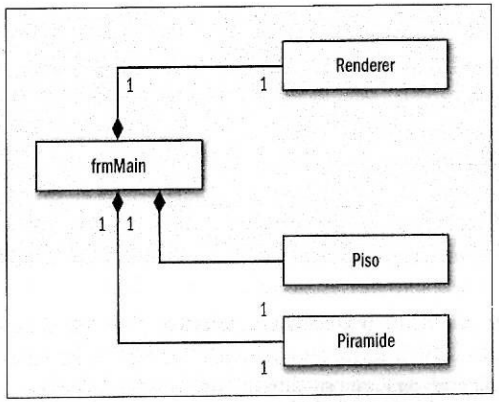


***Figura 2.** Nuestro sencillo diagrama de clases.*

La clase **Renderer** será la encargada de inicializar **Direct3D**, establecer ciertas pautas para el manejo de las matrices del API y, principalmente, mantener alejadas el resto de las clases de código específico de **DirectX**; por lo tanto, el día de mañana podríamos realizar un cambio a otra librería, como **OpenGL**, sin por esto tener que modificar cada vez todas las clases de nuestra aplicación.

La clase **Piso** y la clase **Piramide** representarán los objetos 3D que veremos en pantalla. Cada una de ellas almacenará la información de geometría de los objetos y se comunicará con la clase **Renderer** para transferírselos en el momento de tener que colocarlos en pantalla.
La clase **frmMain** será la principal de nuestra pequeña aplicación: coordinará el trabajo entre las otras clases y mantendrá el bucle en el cual se invocarán los métodos necesarios de las otras clases para dibujar la escena en pantalla.

INICIALIZAR DIRECT3D

El método **Init** será el encargado de inicializar el API para poder comenzar a dibujar objetos. Para esto necesitamos hacer uso principalmente de la clase **Device** de **Direct3D** (espacio de nombres **Microsoft.DirectX.Direct3D**), que es el dispositivo de renderización, y nos permitirá por medio de sus métodos controlar la iluminación, texturado, sombreado y dibujado de los objetos que coloquemos en la escena.

Como primer paso, nos disponemos a realizar la creación de un objeto del tipo **Device** por medio del operador **new** y haciendo uso del constructor de la clase que posee parámetros. Analicemos a continuación el prototipo del método:

```
public Device(
    int adaptador,
    DeviceType tipoDispositivo,
    Control ventanaDeRenderización,
    CreateFlags FlagsDeComportamiento,
    PresentParameters parámetrosDePresentación
);
```

Con el primer parámetro estaremos especificando para qué adaptador deseamos crear el dispositivo. Recordemos que el sistema operativo **Microsoft Windows** puede manejar varios adaptadores de video simultáneamente (en pocas palabras, podríamos tener más de un monitor conectado a la misma PC). Si en este parámetro colocamos un cero, el objeto asumirá que deseamos crear el dispositivo en el adaptador predefinido, que es el único o el principal.
Con el segundo parámetro especificaremos el tipo de dispositivo que deseamos crear. Podremos optar entre las siguientes opciones:

- **DeviceType.Hardware**: se creará un dispositivo de rasterización vía hardware.
- **DeviceType.Reference**: se creará un dispositivo de rasterización vía software, pero el dispositivo hará uso de instrucciones extendidas de CPU cuando éstas se encuentren disponibles.
- **DeviceType.Software**: se creará un dispositivo de rasterización vía software.

Lo ideal es, antes de seleccionar una opción que tenga posibilidades de ser rechazada, llevar a cabo una verificación por medio de algún método de la clase **Manager** del espacio de nombres **Microsoft.DirectX.Direct3D** o, en su defecto, atrapar la excepción y volver a intentar la llamada con otra opción.

El tercer parámetro indica cuál será la ventana (o control) a la que pertenecerá el dispositivo y dónde se realizará el dibujado. En nuestro caso, aquí determinaremos un control del tipo **Panel**, pero podríamos haber especificado la ventana entera en la cual estamos trabajando.

En el cuarto parámetro precisaremos distintas opciones en la creación del dispositivo; algunas de las cuales son mutuamente excluyentes y otras, por el contrario, son combinables. En la página siguiente, analizamos las opciones más importantes:

- **CreateFlags.HardwareVertexProcessing**: posee la función de especificar el procesamiento de vértices por medio del hardware.
- **CreateFlags.SoftwareVertexProcessing**: es la encargarda de especificar el procesamiento de vértices por medio del software.
- **CreateFlags.MixedVertexProcessing**: especifica el procesamiento de vértices por medio de hardware y software de modo combinado.
- **CreateFlags.FpuPreserve**: indica qué tipo de precisión debe usar el dispositivo: simple o doble. De este modo, seleccionaremos doble precisión, que reducirá la performance de **Direct3D**, para una mayor exactitud en cálculos de procesamiento.

Las opciones **HardwareVertexProcessing**, **SoftwareVertexProcessing** y **MixedVertexProcessing** son mutuamente excluyentes y, nuevamente, es posible que la elección de una de ellas cause un fracaso en la creación del dispositivo por falta de soporte a dicha opción en la plataforma de trabajo.
Por último, el quinto parámetro especifica los parámetros de presentación. Esta opción es, a su vez, una clase con una serie de propiedades que deberemos completar antes de invocar el constructor de la clase **Device**. Veamos cómo hacerlo:

```
PresentParameters d3dPresentParams = new PresentParameters();
d3dPresentParams.Windowed = true;
d3dPresentParams.SwapEffect = SwapEffect.Discard;
```

En la primera línea creamos un objeto del tipo **PresentParameters**. Las propiedades especificadas en el listado anterior son las que no deseamos dejar en su valor predefinido:

Windowed: indica si inicializamos el dispositivo en modo ventana o en pantalla completa. Esta opción es muy importante, ya que el modo ventana implica una serie de restricciones en la determinación de otros parámetros. Por ejemplo, no podremos cambiar la resolución al haber otras aplicaciones en el mismo escritorio.
SwapEffect: existe un concepto básico que utilizamos cuando dibujamos **en pantalla**, y es que lo estamos haciendo en una superficie de memoria, que puede ser de

ELECCIÓN DEL TIPO DE DISPOSITIVO

Existen configuraciones más convenientes que otras respecto al tipo de dispositivo por elegir. El problema es que no todo el hardware existente soporta las mismas opciones. En ocasiones no quedará otra opción que procesar vértices por software si deseamos que nuestra aplicación funcione.

video o convencional, y luego, cuando presentamos la información en pantalla, esa superficie es copiada a la memoria de video que está mapeada con lo que refleja el monitor o, en caso de ser memoria de video, se modifica el puntero que indica el comienzo del mapeo. La superficie que no vemos, sobre la que dibujamos, es el **backbuffer**, y la reflejada en pantalla se llama **frontbuffer**. Cuando **Direct3D** realiza el intercambio entre el **backbuffer** y el **frontbuffer**, podemos indicarle que actúe de distintos modos. Nosotros seleccionaremos la opción **SwapEffect.Discard**, ya que no deseamos preservar la información existente en el buffer sobrescrito.

El resto de las propiedades de **PresentParameters** podremos dejarlas en su valor predeterminado. Cuando avancemos en el desarrollo de nuestro ejemplo, deberemos modificar otras propiedades de su valor predefinido.

Veamos cómo queda hasta ahora el método **Init** de la clase **Renderer** que creamos:

```
public bool Init(Control devWin)
{
    PresentParameters d3dPresentParams = new PresentParameters();
    d3dPresentParams.Windowed = true;
    d3dPresentParams.SwapEffect = SwapEffect.Discard;
    try
    {
        m_device = new Device(0, DeviceType.Hardware, devWin,
            CreateFlags.HardwareVertexProcessing, d3dPresentParams);
    }
    catch (DirectXException)
    {
        try
        {
            m_device = new Device(0, DeviceType.Hardware, devWin,
                CreateFlags.SoftwareVertexProcessing, d3dPresentParams);
        }
        catch (DirectXException)
        {
            return false;
        }

    }
            return DisponerEscena();
}
```


El método **Init** recibe el control sobre el que se creará el dispositivo. Existen dos bloques **try.catch** anidados, ya que en primera instancia intentamos crear un dispositivo que procese vértices por hardware, pero en caso de fracasar, repetimos el intento especificando que el procesamiento de vértices se realice por software.

Por último, el método **Init** realizará una llamada a otro método de la misma clase denominado **DisponerEscena**. Lo que hará principalmente este último método es fijar los valores predeterminados a la matriz de vista y proyección:

```
public bool DisponerEscena()
{
    // Por el momento, no utilizaremos iluminación
    m_device.RenderState.Lighting = false;
      // Fijamos el valor de la matriz de vista por medio
      del método estático de Matrix, LookAtLH
    m_device.Transform.View = Matrix.LookAtLH( new Vector3(0.0f,
      50.0f, -50.0f),
      new Vector3(0.0f, 0.0f, 0.0f), new Vector3( 0.0f, 1.0f, 0.0f ) );
    // Fijamos el valor de la matriz de proyección, en perspectiva,
      por medio del método estático de Matrix, PerspectiveFovLH
    m_device.Transform.Projection = Matrix.PerspectiveFovLH( (float)
      Math.PI / 4, 1.0f, -30.0f, 30.0f );
    return true;
}
```

El método consiste básicamente en tres pasos:

1. Establecemos, por el momento, que nuestra escena no posea iluminación por medio de la propiedad **RenderState.Lighting** de la clase **Device**. La propiedad **RenderState** nos será muy útil en el futuro porque modifica opciones internas de **Direct3D** que alteran el modo en que son dibujados los objetos en pantalla.
2. Fijamos el valor de la matriz de vista haciendo uso del método estático **LookAtLH** de la clase **Matrix**, que nos permite especificar, por medio de tres parámetros, desde qué punto deseamos mirar la escena, a qué punto deseamos mirar y, finalmente, cuál es el "arriba" en nuestro mundo.
3. Fijamos el valor de la matriz de proyección por medio del método estático **PerspectiveFovLH** de la clase **Matrix** por medio de cuatro parámetros: campo de visión (especificamos lo usual, que es el número PI dividido cuatro), la relación existente entre el alto y ancho de nuestra escena, el plano de recorte cercano y el plano de recorte lejano. Todas estas propiedades delimitan el volumen de visualización de la escena que observaremos.

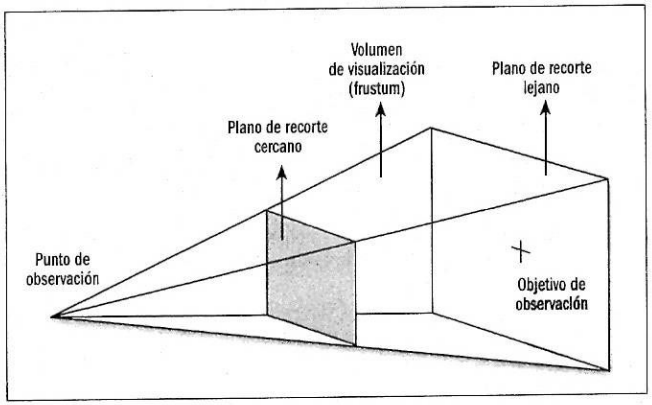


Figura 3. *El volumen de visualización.*

Con nuestro flamante método **Init**, veremos ahora cómo hacer uso de él por medio de una instancia de **Renderer** en la clase **frmMain**:

```
static void Main()
{
    // Instancio un nuevo objeto
    frmMain frm = new frmMain();
    // Muestro el formulario
    frm.Show();
    // Arranco la aplicación

    frm.Arrancar();
    // Mientras que el formulario no sea destruido
    while(frm.Created)
    {
        // Dibujo la escena
        frm.Dibujar();
        // Atiendo eventos
        Application.DoEvents();
    }
}
```

Analicemos el código anterior:

En la primera línea, lo que hicimos fue crear una instancia del formulario **frmMain**. Por medio del método **Show**, lo mostramos en pantalla; luego invocamos un método llamado **Arrancar** creado por nosotros y donde simplemente realizaremos la llamada al método **Init** de **Renderer**. Tengamos en cuenta que la clase **frmMain** tendrá una propiedad de este tipo.

Finalmente ingresamos en un bucle, donde, mientras el formulario no sea destruido, invocaremos el método **Dibujar** (que fue generado por nosotros y que ahora veremos qué contiene) y luego el método estático **DoEvents**, de la clase **Application**, para que nuestra aplicación se encuentre posibilitada de seguir atendiendo mensajes externos a pesar de encontrarse en un bucle.

Por lo tanto, podríamos expresar el diagrama del método **Main** del siguiente modo:

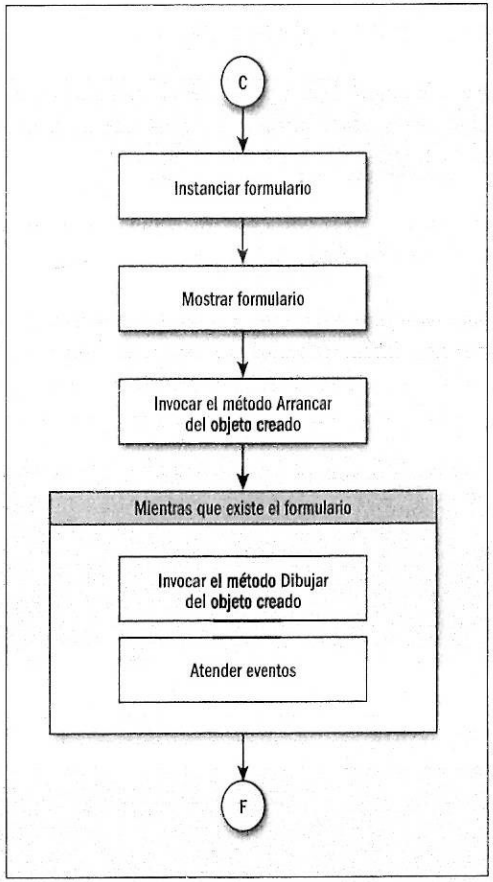


Figura 4. *Mientras la aplicación se encuentre en ejecución, permaneceremos dentro del bucle.*

Prestemos especial atención a que el comportamiento de nuestra aplicación es distinto del de una aplicación convencional, ya que aquí estaremos dentro de un ciclo dibujando una y otra vez, en contraposición a la aplicación típica que sólo redibuja atendiendo el clásico mensaje **WM_PAINT** que se suscita cuando existe una actualización en los datos (a pedido nuestro) o cuando la ventana es sobrescrita por algún elemento que se mueva por encima de ella.

Este modo de trabajar es muy común en los juegos de computadora, y a dicho ciclo se lo denomina bucle del juego (**game loop**), donde también se suelen realizar otras tareas, como atender los dispositivos de entrada, procesar información referida a personajes manejados por la máquina, etc.

Pero ¿qué hacemos en el método **Dibujar**? Veamos:

El método **Dibujar** se encargará de invocar un método homónimo para cada uno de los objetos contenidos en la escena (la pirámide y el piso), además de invocar los métodos correspondientes de **Renderer** para que todo esté listo y en forma para el dibujado.

Observemos un diagrama de secuencia para ver cómo trabajan los distintos objetos de nuestro pequeño sistema (**Figura 5**).

El método **Limpiar** de **Renderer** simplemente invocará el método **Clear** de la clase **Device**, que puede ser utilizado de varios modos; por el momento sólo lo emplearemos para limpiar la superficie donde estamos dibujando y especificaremos por medio del segundo parámetro el color con el que deseamos limpiarla.

```
public void Limpiar()
{
    m_device.Clear(ClearFlags.Target, System.Drawing.Color.Black,
        1.0f, 0);
}
```

FOROS DE AYUDA

En el sitio **www.dedalus-software.com.ar** podrán encontrar foros específicos de programación C# en donde se trata acerca de Managed DirectX.

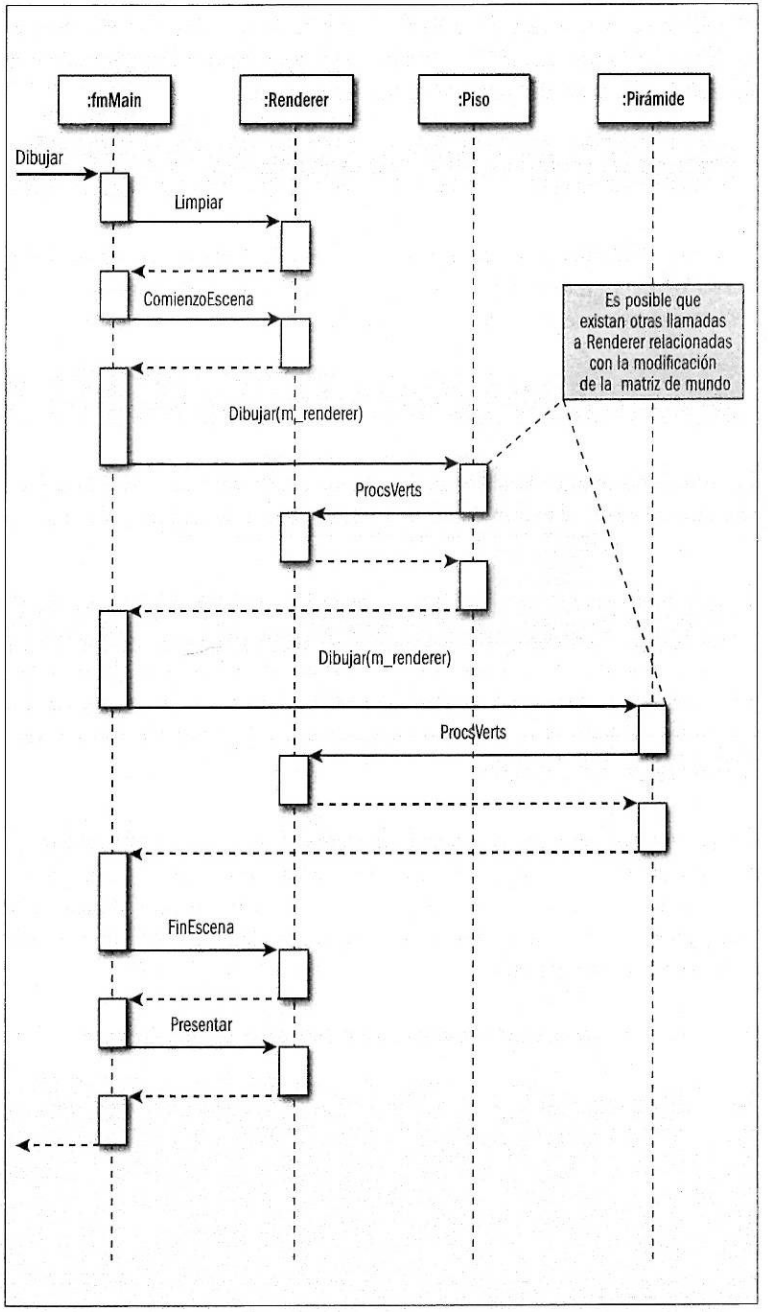


Figura 5. *Cada objeto dibujable posee su propio método Dibujar.*

El método **ComienzoEscena** de **Renderer** se ocupará de invocar el método **BeginScene** de la clase **Device**. **Direct3D** nos indica que se debe llamar a este método previamente a la realización del dibujado de cualquier objeto.

```
public void ComienzoEscena() { m_device.BeginScene();}
```

El método **FinEscena** de **Renderer** invocará el método **EndScene** de la clase **Device**. **Direct3D** indica que se debe llamar este método cuando finalizamos el procesamiento de vértices relacionados con una escena.

```
public void FinEscena() { m_device.EndScene(); }
```

El método **Presentar** de **Renderer** invocará el método **Present** de la clase **Device**. Dicho método vuelca el contenido del **backbuffer** sobre el **frontbuffer**, refrescando lo que vemos en pantalla.

```
public void Presentar() { m_device.Present();}
```

DIBUJAR OBJETOS

Por cada objeto mostrado en pantalla deberemos poseer un array de vértices, del formato seleccionado, con la información de geometría de éste. Por esta razón, es conveniente crear una clase para cada uno de ellos. Luego, existirá un método llamado **Dibujar** en el cual le pasaremos a la clase **Renderer** el contenido de dicho array para que sea procesado.

Así, pues, crearemos la clase **Piso** con el array de vértices correspondiente:

```
public class Piso
{
    // Array de vértices del piso
private CustomVertex.PositionColored[] m_verts =
  new CustomVertex.PositionColored[6];
    // ...
};
```


Nuestros vértices serán del tipo **CustomVertex.PositionColored**, ya que esta estructura tiene las propiedades que requerimos (posición + color). Por lo tanto, llevaremos a cabo la creación de un array de seis elementos, pues nuestro piso estará formado por dos triángulos compuestos del siguiente modo:

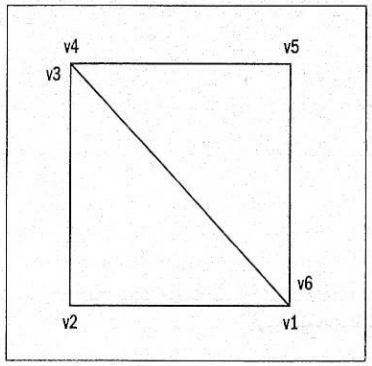


***Figura 6.** Nuestro piso estará formado por dos triángulos.*

Como la información del piso no variará, lo único que tendremos que rellenar será este array una vez. Esto no quiere decir que el piso no pueda ser movido, escalado o rotado en cualquier eje, porque para eso podremos modificar la matriz de mundo, que rectificará a su vez el vértice entrante para entonces colocarlo en la posición que nos plazca. En consecuencia, podremos completar la información del array en el constructor de la clase **Piso**:

```
public Piso()
{
    // Cara A
    // Vértice 1
    m_verts[0].X = 25.0f; m_verts[0].Y = -3.0f; m_verts[0].Z = -25.0f;

    m_verts[0].Color = System.Drawing.Color.SkyBlue.ToArgb();
    // Vértice 2
    m_verts[1].X = -25.0f; m_verts[1].Y = -3.0f; m_verts[1].Z = -25.0f;
    m_verts[1].Color = System.Drawing.Color.SlateBlue.ToArgb();
    // Vértice 3
    m_verts[2].X = -25.0f; m_verts[2].Y = -3.0f; m_verts[2].Z = 25.0f;
    m_verts[2].Color = System.Drawing.Color.SkyBlue.ToArgb();
    // Cara B
    // Vértice 1
```

```
        m_verts[3].X = -25.0f; m_verts[3].Y = -3.0f; m_verts[3].Z = 25.0f;
        m_verts[3].Color = System.Drawing.Color.SkyBlue.ToArgb();
        // Vértice 2
        m_verts[4].X = 25.0f; m_verts[4].Y = -3.0f; m_verts[4].Z = 25.0f;
        m_verts[4].Color = System.Drawing.Color.SlateBlue.ToArgb();
        // Vértice 3
        m_verts[5].X = 25.0f; m_verts[5].Y = -3.0f; m_verts[5].Z = -25.0f;
        m_verts[5].Color = System.Drawing.Color.SkyBlue.ToArgb();
}
```

Tal como podemos apreciar, cada uno de los vértices posee una posición y un color. Si especificamos distintos colores para vértices de un mismo triángulo, **Direct3D** se ocupará de realizar una interpolación de éstos para otorgarles color a los pixeles intermedios en el proceso de rasterización.

Ahora veamos el método **Dibujar** de la misma clase:

```
public void Dibujar(Renderer rnd)
{
    rnd.SetMatrixMode(eMatrixMode.MM_WORLD);
    rnd.LoadIdentity();
            rnd.ProcsVerts(m_verts, 6);
}
```

Bastante sencillo, simplemente hemos modificado la matriz de mundo y la llevamos a su valor identidad (una matriz identidad consiste en un elemento neutro en la operación producto) para tener la certeza de que no exista un valor distinto al deseado que haya quedado por el dibujado de otro objeto previo y termine transformando nuestro piso involuntariamente.

III OTROS COMPONENTES DE DIRECTX

DirectX no es sólo Direct3D. Con esta API, también podremos acceder a dispositivos de entrada de un modo eficiente (DirectInput), reproducir sonido (DirectSound y DirectMusic), reproducir video (DirectShow), etc.

Por último, invocamos el método **ProcsVerts** de la clase **Renderer** y le pasamos el array junto con la cantidad de elementos que posee.
La clase **Piramide** es muy similar a **Piso**, sólo que para variar un poco el ejemplo le colocamos algunas propiedades más que permitan su rotación en eje Y y su escalamiento. El método **Dibujar** es similar al de la clase **Piso**, pero con unas líneas más.

```
public void Dibujar(Renderer rnd)
{
    rnd.SetMatrixMode(eMatrixMode.MM_WORLD);
    rnd.LoadIdentity();
            // Modifico la escala |
    if (m_fEscalaX != 1.0f || m_fEscalaY != 1.0f || m_fEscalaZ != 1.0f)
        rnd.Scale(m_fEscalaX, m_fEscalaY, m_fEscalaZ);
    // Modifico el ángulo de rotación sobre el eje Y
    if (m_fAngleY != 0.0f)
        rnd.RotateY(m_fAngleY);
    rnd.ProcsVerts(m_verts, 12);
}
```

En primera instancia modificamos la matriz de mundo y la llevamos a su valor identidad. Luego, si la propiedad **m_fEscalaX** o **m_fEscalaY** o **m_fEscalaZ** es distinta de uno, invocamos el método **Scale** de **Renderer**.
De modo análogo, si la propiedad **m_fAngY** es distinta de cero, invocamos el método **RotateY** de la clase **Renderer**.

Para concluir, invocamos el método **ProcsVerts** para procesar los vértices.

El método ProcsVerts de la clase Renderer

```
public void ProcsVerts(CustomVertex.PositionColored[] verts, int iVertCount)
{
    m_device.VertexFormat = CustomVertex.PositionColored.Format;
m_device.DrawUserPrimitives(PrimitiveType.TriangleList,
   iVertCount / 3, verts);
}
```

El método **ProcsVerts**, que se encuentra incluido en la clase **Renderer**, tiene la función de procesar los vértices recibidos. Para ello, actualizaremos la propiedad

VertexFormat de la clase **Device** al formato del vértice que nos disponemos a procesar. Luego, invocamos el método **DrawUserPrimitives**. En el primer parámetro especificamos el tipo de primitiva, en el segundo parámetro detallamos qué cantidad de primitivas hay que realizar y, por último, pasamos el array donde se encuentran los vértices de nuestro objeto.

Los tipos de primitivas

Tal como lo habíamos mencionado anteriormente, **Direct3D** sólo sabe de triángulos; esto significa que todo objeto que deseemos dibujar deberá estar formado por esta figura geométrica básica.

Sin embargo, es posible indicarle que a partir de un grupo de vértices dibuje también puntos y líneas con distintos criterios. Veamos cuáles son estas primitivas:

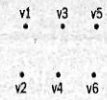
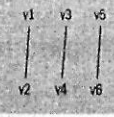
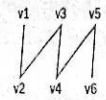
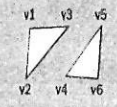
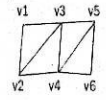
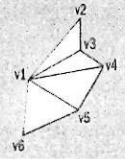
PRIMITIVA	DESCRIPCIÓN	EJEMPLO
PointList	Lista de puntos, Direct3D tomará cada vértice del grupo y dibujará un punto en su posición.	
LineList	Lista de líneas, Direct3D tomará de a dos vértices y los utilizará para trazar segmentos independientes.	
LineStrip	Tira de líneas, Direct3D tomará los dos primeros vértices del grupo y trazará un segmento, luego tomará el siguiente elemento y trazará otro segmento conectado con el anterior, y así sucesivamente. De este modo queda formada una sucesión de segmentos conectados entre sí.	
TriangleList	Lista de triángulos, Direct3D tomará de a tres vértices formando triángulos independientes.	
TriangleStrip	Tira de triángulos, Direct3D tomará los primeros tres vértices y formará un triángulo, luego tomará el siguiente vértice y, haciendo uso de los dos anteriores formará otro triángulo, y así sucesivamente.	
TriangleFan	Fan de triángulos, Direct3D tomará el primer vértice y otros dos con los que formará un triángulo, luego tomará otros dos y junto con el primero creará otro triángulo, y así sucesivamente.	

***Tabla 1.** Tipos de primitivas.*

EJECUTAR LA APLICACIÓN

La escena puede ser rotada en los ejes **X** e **Y** por medio del uso del mouse, y también estaremos en condiciones de modificar la escala de la pirámide en cualquiera de sus componentes y hacerla rotar sobre su eje **Y**. Todo esto, sencillamente, manipulando las matrices de mundo y vista.

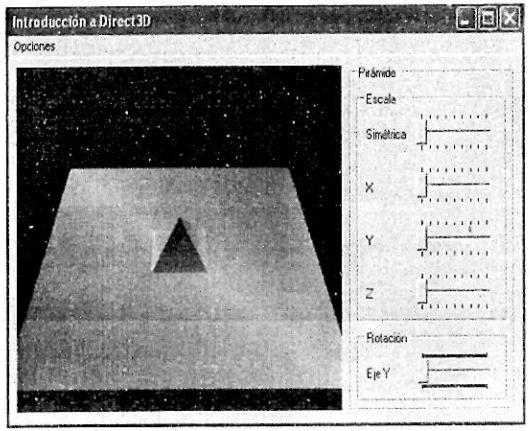


***Figura 7.** Nuestra aplicación en acción.*

En el sitio web **onweb.tectimes.com**, se encuentra disponible para su descarga la aplicación de ejemplo que acompaña este capítulo.

RESUMEN

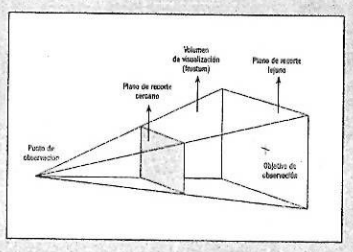
Direct3D es una API extensa y compleja, pero las posibilidades que ofrece son prácticamente ilimitadas. Sus usos también son muy variados, y no sólo se utiliza en juegos de computadora, sino también en aplicaciones de medicina, visualización de terrenos, gráficos estadísticos dinámicos en 3D, etc.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Se puede modificar la posición de un objeto en pantalla sin variar la posición de sus vértices?

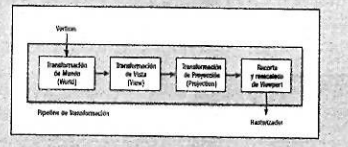
2 ¿Cómo es posible que un vértice de tres dimensiones se multiplique a una matriz de 4x4?



3 ¿Para qué me convendría descartar vértices no visibles? ¿Acaso el pipeline no los termina recortando en el proceso de recorte?

4 ¿Cómo pinto un triángulo Direct3D si éste posee colores diferentes en cada uno de sus vértices?

5 ¿Es posible modificar las operaciones que se suceden dentro del pipeline?



Apéndice A

Documentación de código

La documentación del código de un sistema (en un formato práctico) es una tarea ardua y tiende a quedar obsoleta con el devenir de nuevas versiones. Pero el código de un programa debería estar debidamente comentado para facilitar el trabajo del programador y evitar los errores derivados de una mala interpretación de lo que debería hacer una función.

CÓMO DOCUMENTAR CÓDIGO

Tanto fastidio suele acarrear esta labor, que muchas veces no se realiza, o se hace sólo en la primera versión, o se efectúa parcialmente con las clases más importantes del sistema. Por suerte, existen herramientas que facilitan tal tarea.

Básicamente, toman la información de los comentarios que insertamos en el código (con un formato especial), la procesan y generan un documento que puede abrirse con alguna aplicación de uso estándar. El compilador C# provee una de esas herramientas y genera documentación **XML** de nuestro código.

Veamos un ejemplo de nuestro código:

```
/// <summary>
/// Descripción del namespace ProgCS
/// </summary>
namespace ProgCS
{
    /// <summary>
    /// Descripción de la clase Class1
    /// </summary>
    ///
    class Class1
    {

        /// <summary>
        /// Descripción del método m1
        /// </summary>
        public void m1()
        {
        }
        /// <summary>
        /// Descripción del método m2
        /// </summary>
        public void m2()
        {
        }
        /// <summary>
        /// Descripción del método m3
        /// </summary>
```



```
        /// <param name="num">Descripción parámetro num</param>
        /// <returns>Descripción del retorno del método</returns>
        public bool m3(int num)
        {
            return true;
        }
    }
}
```

Si compilamos nuestro código en comando de línea, usando directamente el compilador (**csc.exe**), entonces deberemos aplicar el switch **/doc** del siguiente modo:

```
csc otras_opciones /doc:nombre_archivo
```

Veamos un ejemplo para el código anterior:

```
csc Class1.cs /doc:docxml.xml
```

Hecho esto, el compilador generará (además de la salida usual del programa compilado) un archivo llamado **docxml.xml** (podría llamarse de cualquier otro modo), que para nuestro ejemplo poseerá el siguiente contenido:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Class1</name>
    </assembly>
    <members>
        <member name="T:ProgCS.Class1">
            <summary>
            Descripción del namespace ProgCS </summary>
        </member>
        <member name="M:ProgCS.Class1.m1">
            <summary>
            Descripción del método m1
            </summary>
        </member>
```

```
        <member name="M:ProgCS.Class1.m2">

            <summary>
            Descripción del método m2
            </summary>
        </member>
        <member name="M:ProgCS.Class1.m3(System.Int32)">
            <summary>
            Descripción del método m3
            </summary>
            <param name="num">Descripción parámetro num</param>
            <returns>Descripción del retorno del método</returns>
        </member>
    </members>
</doc>
```

Si trabaja con el excelente **Visual Studio .NET**, no querrá abandonarlo por un segundo. Lo mismo puede especificarse en las propiedades del proyecto, en la sección **Configuration Properties**, subsección **Build**, y allí, en **XML Documentation File**.

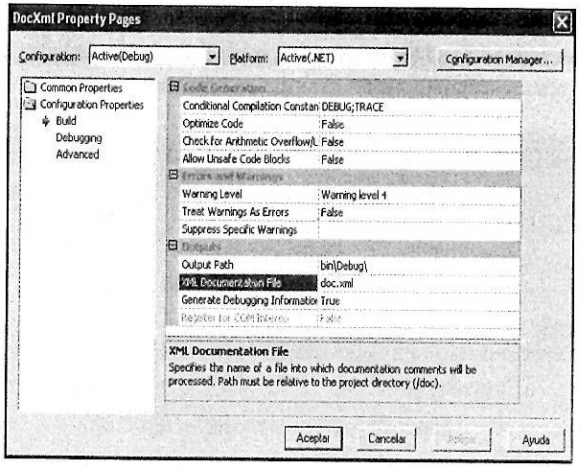


Figura 1. *Propiedades del proyecto.*

Si en lugar de un archivo **XML** deseamos obtener archivos **HTML**, navegables en cualquier browser, es posible utilizar una facilidad de **Visual Studio .NET**. Ésta puede ser accedida desde el menú **Tools**, opción **Build Comment Web Pages** (**Figura 2**).

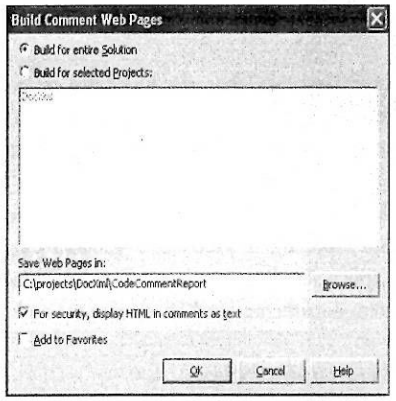


Figura 2. *Creando página de ayuda en formato HTML.*

Presionando el botón **OK**, podremos ver el resultado de la operación en una de las páginas abiertas por el entorno, o bien podremos abrirla con cualquier navegador.

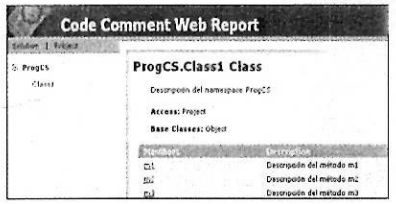


Figura 3. *Una de las páginas generadas por la herramienta.*

Las páginas generadas se encuentran divididas de modo vertical. A la izquierda podemos seleccionar qué espacio de nombres o clase deseamos consultar por información, y a la derecha se nos presenta la información respectiva.

Los tags disponibles

Todos los tags (marcas) dentro de nuestro código deben ser colocados dentro de un comentario con una barra extra:

```
///
```

Veamos algunos de los tags más importantes:

- **<summary>...</summary>**: es uno de los más importantes. Especifica un resumen de la estructura que precede. Por ejemplo:

```
/// <summary>
/// Descripción del método m1
/// </summary>
public void m1()
{
}
```

- **<param = "">...</param>**: permite especificar y describir un parámetro.

```
/// <summary>
/// </summary>
/// <param name="num">Descripción parámetro num</param>
public bool m3(int num)
{
}
```

- **<returns>...</returns>**: permite especificar qué debe devolver un método.

```
/// <summary>
/// </summary>
/// <returns>Descripción del retorno del método</returns>
public bool m4()
{
    return true;
}
```

- **<example>...</example>**: nos permite especificar un ejemplo de uso de la clase, método, variable, etc.
- **<code>...</code>**: nos permite especificar un trozo de código (usualmente este tag se coloca dentro del tag de **example**).

Otras herramientas para crear documentación

Existe otra herramienta muy popular que trabaja de un modo similar al que vimos. Su nombre es **Doxygen** y se puede descargar libremente desde **www.doxygen.org**. **Doxygen** genera documentación en formato **HTML**, **LaTeX**, **RTF**, **PostScript**, **PDF** y **HTML** comprimido (**chm**).

Apéndice B

La notación Húngara

La notación Húngara es una convención de nombramiento de variables y funciones. Es muy utilizada en programación en el lenguaje C++. También tiene una menor incidencia en lenguajes como C# y Java. En este apéndice veremos los conceptos fundamentales.

SERVICIO DE ATENCIÓN AL LECTOR: lectores@mpediciones.com

LA CONVENCIÓN

Al nombrar las variables, dificultamos la lectura del programa. Si a una variable simplemente la llamamos **X**, luego podríamos no recordar su contenido, ocasionando errores o retrasos en la codificación.
Los identificadores de esta notación se dividen en tres partes.

Un tipo base

Corresponde a la variable que será nombrada. No siempre corresponde a un tipo de dato fundamental, sino a uno más abstracto. Debe ser escrito en minúsculas.

TIPO BASE	SIGNIFICADO
s	Short
i, n	Int
l	Long
us	Ushort
ui	Uint
ul	Ulong
c	Char
d	Double
f	Float
e	Enum
b	Bool

***Tabla 1.** Datos de tipo base.*

Un prefijo

Se especifica antes del tipo base y describe el uso de la variable.

PREFIJO	SIGNIFICADO
a	Array
c	Contador
d	Diferencia entre dos variables del mismo tipo
e	Elemento de un array
g_	Variable global
h	Handler (manejador)
i	Índice de un array
m_	Variable miembro
p	Puntero

***Tabla 2.** Prefijos.*

Un calificador

Es la parte descriptiva de la variable o función que completa el nombre. Se recomienda usar nombres que tengan un significado apropiado con cada variable.
Si un calificador está compuesto por más de una palabra, éstas se escribirán juntas, pero se colocará la primera letra de cada palabra en mayúscula (no se permite la separación de palabras con guiones de ningún tipo).

Lista de calificadores estándar

Debido a que cierto tipo de variables aparecen en una gran cantidad de programas, se recomienda utilizar los nombres sugeridos por la notación.

CALIFICADOR	SIGNIFICADO
min	El elemento más pequeño de una lista o array
first	El primer elemento de una lista o array
last	El último elemento de una lista o array
lim	Un número límite o umbral
max	El elemento más grande de una lista o array
i, j, k, m, n	Contadores

***Tabla 3.** Calificadores estándar.*

ALGUNOS EJEMPLOS

Veamos, finalmente, algunos ejemplos de nombramientos:

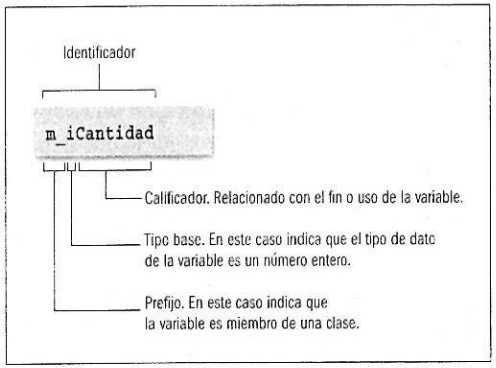


***Figura 1.** Ejemplo de nombramientos.*

¿Cómo nombraríamos una variable que contuviera:

- la edad de una persona (variable local)?: **iEdad** (**i** es el tipo base, y **Edad**, el calificador; no posee prefijo).
- la edad de una persona (propiedad de una clase)?: **m_iEdad** (**m_** es el prefijo que indica que la variable es miembro de una clase, **i** es el tipo base y **Edad** es el calificador).
- el resultado de una operación (variable local)?: **bRes** (**b** es el tipo base [bool] y **Res** es el calificador que abrevia la palabra **resultado**; no posee prefijo).
- un array de números flotantes que correspondiese a una lista de medidas (propiedad de una clase)?: **m_afMedidas** (**m_a** es el prefijo [propiedad de una clase y de tipo array], **f** es el tipo base [float] y **Medidas** es el calificador).
- el número más pequeño que se obtiene en un bucle?: **min** (nombre estándar).
- un contador utilizado dentro de una sentencia **for**?: **i** (nombre estándar).

VENTAJAS Y DESVENTAJAS DE LA CONVENCIÓN

Ahora, podríamos estar preguntándonos por qué deberíamos utilizar esta convención. En primer lugar es muy recomendable adoptar alguna convención ampliamente utilizada, ya que de participar en un proyecto de construcción de software integrado por muchas personas, leer código ajeno será mucho más sencillo.

Ventajas propias de la notación Húngara:

- Permite conocer el tipo de variable solamente viendo el identificador.
- Los nombres fijados con esta convención quedan relativamente cortos.
- Es ampliamente utilizada.

Quienes no gustan de esta notación esgrimen principalmente dos motivos:

- El identificador queda ligado al tipo de dato; por lo que si queremos cambiar el tipo de la variable, deberemos cambiar su nombre en todas las ocurrencias.
- Los identificadores, en algunos casos, son difíciles de leer a primera vista.

La razón por la cual la convención en lenguajes como C# y Java no es tan popular radica principalmente en la evolución de los entornos de desarrollo integrados que ofrecen la información de tipos de datos y características de una variable simplemente con posar el puntero del mouse sobre ellas.
Sin embargo, en código impreso, el entorno de desarrollo no podría ser utilizado, y en el análisis de una gran porción de código por medio de la notación Húngara identificaríamos las características de variables de manera inmediata.

Apéndice C

UML

El lenguaje de modelado unificado, también conocido como UML, es un sistema notacional destinado al modelado de aplicaciones que se basen, principalmente, en conceptos de la programación orientada a objetos.

INTRODUCCIÓN

El UML se encuentra compuesto por una serie de diagramas que nos ayudarán a diseñar y documentar los diferentes componentes del sistema.
Muchos programadores comienzan sus proyectos escribiendo código, pero en proyectos de gran complejidad, esto es un error, sobre todo si existe un equipo de trabajo donde muchos programadores deben trabajar en distintas piezas de un rompecabezas. ¿Qué ocurriría si luego de meses de ardua labor las piezas no se conectaran correctamente? Sería necesario un análisis y un diseño: UML es la herramienta para ello.
Veamos cada uno de los diagramas y los elementos que componen este sistema.

Diagramas de casos de uso

Es una visión de la comunicación usuario/sistema. Es más útil en sistemas que posean un flujo de trabajo, donde distintos usuarios (actores) acceden al sistema desde distintas interfaces. Los casos de uso describen la funcionalidad de un sistema y su utilización por parte de los usuarios. Contienen los siguientes elementos:

- **Actores**: representan a los usuarios del sistema (humanos u otros sistemas).
- **Caso de uso**: representa una determina funcionalidad provista por el sistema a los usuarios.

Estos diagramas son muy utilizados en sistemas basados en flujos de trabajo.

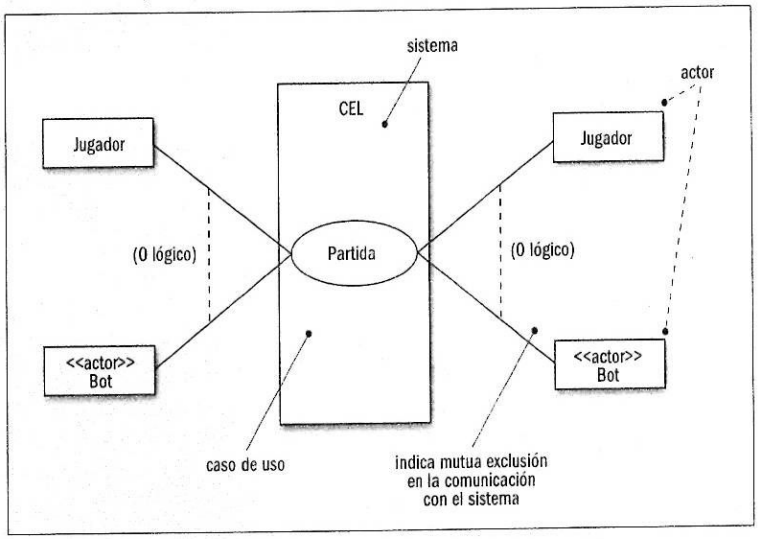


Figura 1. *Diagrama de casos de uso.*

Diagramas de clases

Describen la estructura estática de un sistema, cómo se encuentra estructurado más que cómo se comporta. Contiene los siguientes elementos:

- **Clases**: que representan entidades con características comunes (atributos, operaciones y asociaciones).
- **Asociaciones**: que representan relaciones entre dos o más clases.

Es uno de los más populares. Posee distintos niveles de detalles (mostrando o no atributos y operaciones).

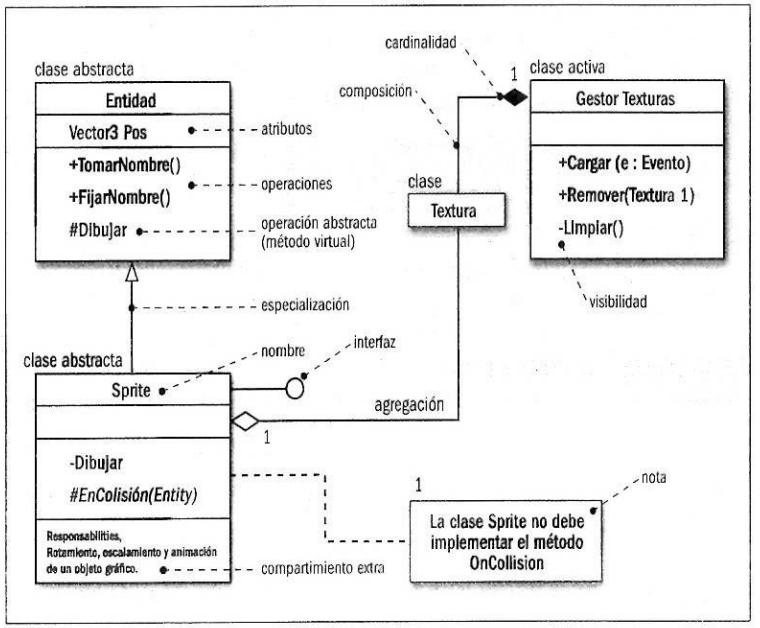


Figura 2. *Diagrama de clases.*

Diagramas de objetos

Describen la estructura de un sistema en un determinado momento; el diagrama de clases lo hace en todo tipo de situación (ya que no es posible crear clases de modo dinámico), pero la cantidad de objetos de nuestro sistema puede cambiar durante la ejecución.
Contiene los siguientes elementos:

- **Objetos**: representan entidades en particular. Son instancias de clases.

- **Enlaces**: representan relaciones entre objetos. Son instancias de asociaciones.

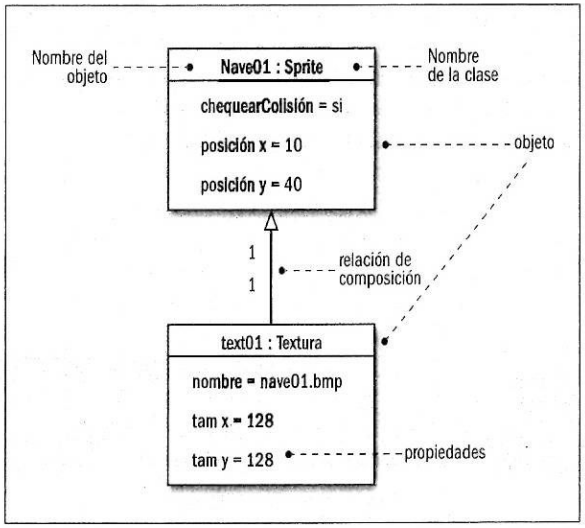


***Figura 3.** Diagrama de objetos.*

Diagramas de secuencia

Describen interacciones entre objetos. Estas interacciones se modelan como intercambio de mensajes. Estos diagramas se focalizan en los mensajes que se intercambian para mostrar determinado comportamiento o acción.

Contiene los siguientes elementos:

- **Roles de clases**: representan las funciones que los objetos asumen en determinadas interacciones.
- **Líneas de vida**: representan la existencia de un objeto durante un período de tiempo.
- **Activaciones**: representan el tiempo durante el cual un objeto se encuentra realizando una acción específica.
- **Mensajes**: representan la comunicación entre dos objetos.

Son especialmente útiles para especificar la precedencia necesaria en la invocación de métodos de unos objetos en particular.

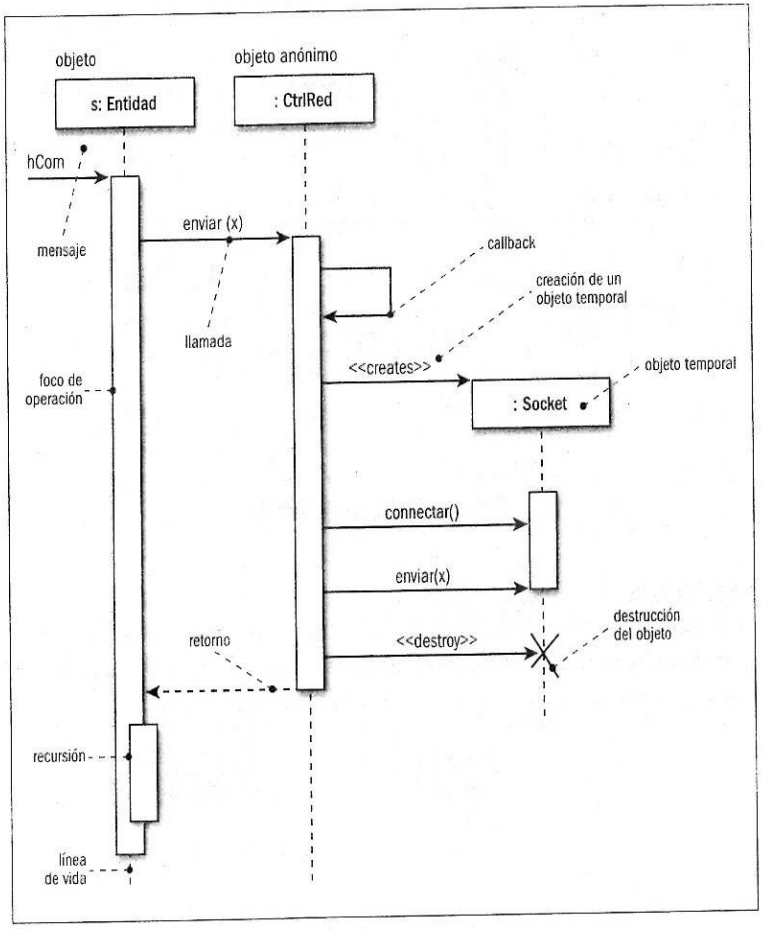


***Figura 4.** Diagrama de secuencia.*

Diagramas de estados

Los objetos poseen un estado, y es posible que la acción que ejecutarán en función de estímulos externos se base en el estado en el que se encuentren. El diagrama de estados se encarga de describir cuáles son los posibles para un determinado objeto y de qué modo se pasa de uno a otro. Está compuesto por los siguientes elementos:

- **Estados**: representan situaciones durante el tiempo de vida de un objeto cuando se cumplen ciertas condiciones, se realiza o se espera la ocurrencia de un evento.
- **Transiciones**: representan relaciones entre los diferentes estados de un objeto.

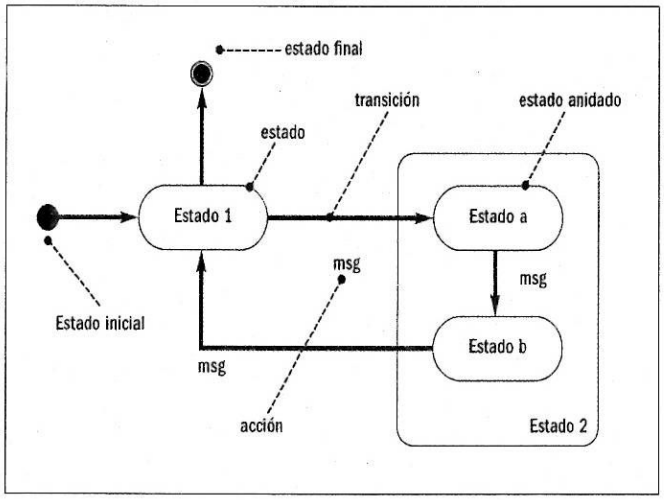


Figura 5. *Diagrama de estados.*

Diagramas de colaboración

Describen la interacción entre clases y asociaciones. Estas interacciones son modeladas como intercambios de mensajes entre clases a través de sus asociaciones.

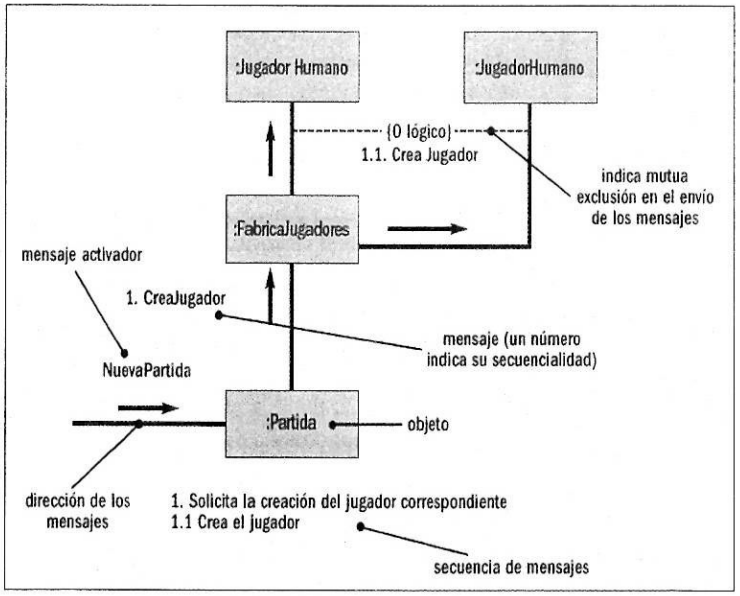


Figura 6. *Diagrama de colaboración.*

Contiene los siguientes elementos:

- **Roles de clases**: representan las funciones que asumen los objetos.
- **Roles de asociación**: representan las funciones que los enlaces deben asumir en una interacción.
- **Flujo de mensajes**: representan los mensajes enviados por objetos a través de enlaces. Los enlaces implementan la distribución de mensajes.

Diagramas de actividad

Describen la actividad de una clase determinada. Estos diagramas son similares a los de estado y utilizan una convención similar, sólo que describen el comportamiento de una clase en respuesta a procesos internos, y no a eventos externos.

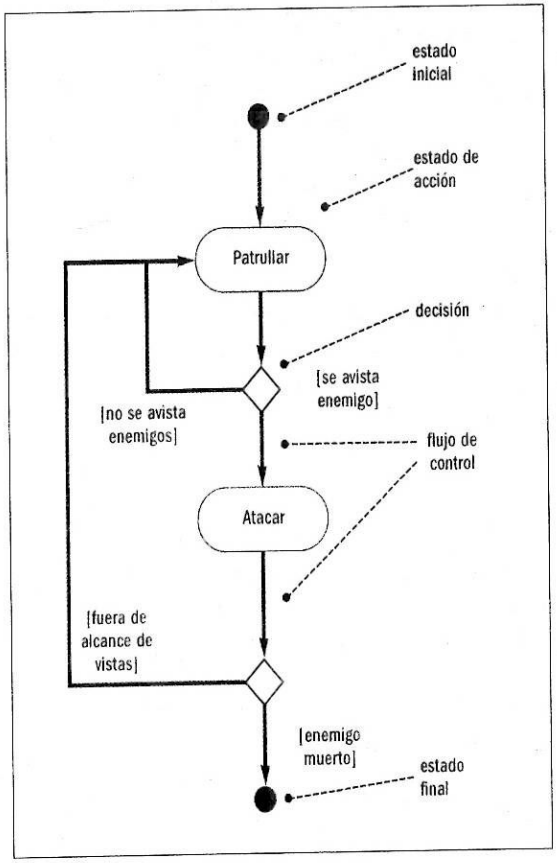


Figura 7. *Diagrama de actividad.*

- **Estados de acción**: representan acciones atómicas (o no interrumpibles), pasos en la ejecución de un algoritmo.
- **Flujos de acciones**: representan relaciones entre diferentes estados de acción de una entidad.
- **Flujos de objetos**: representan la utilización de objetos por estados de acción y la influencia de éstos en otros objetos.

Diagramas de componentes

Describen la organización y dependencia entre los distintos componentes de software en una implementación. Sus componentes representan unidades físicas distribuibles, incluyendo código fuente, código objeto y código ejecutable.

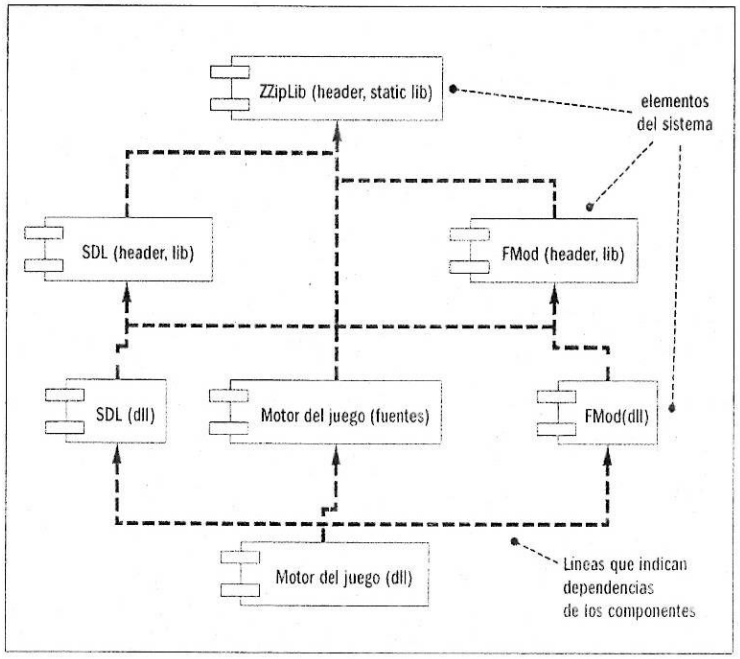


***Figura 8.** Diagrama de componentes.*

Extensiones a UML

Un diagrama puede ser incapaz de expresar una cierta arista de nuestro sistema. Por ello UML posee un sistema de extensión y personalización con el cual se puede agregar nueva simbología a los diagramas existentes, para que se adapten a nuestras necesidades y poder usar nuevos estereotipos, propiedades, valores y restricciones.

PROGRAMACIÓN C#

Servicios al lector

En esta sección encontrará una completa guía de sitios que contienen compiladores para la plataforma .NET. Además, el índice temático, que le permitirá acceder al contenido de este manual de forma precisa y rápida.

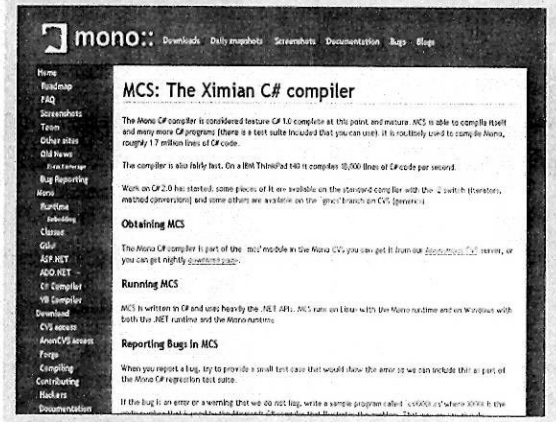
SERVICIO DE ATENCIÓN AL LECTOR: lectores@mpediciones.com

LENGUAJES .NET

En la siguiente lista se detallan compiladores específicos que generan código en **MSIL** para poder ser ejecutados sobre plataforma **.NET**. En algunos casos son lenguajes nuevos, y en otros, compiladores basados en conocidos lenguajes que anteceden a **.NET**.

Como podrá apreciar, no son pocos los emprendimientos que se sumaron a la movida **.NET**, y cada día aparecen nuevos proyectos.

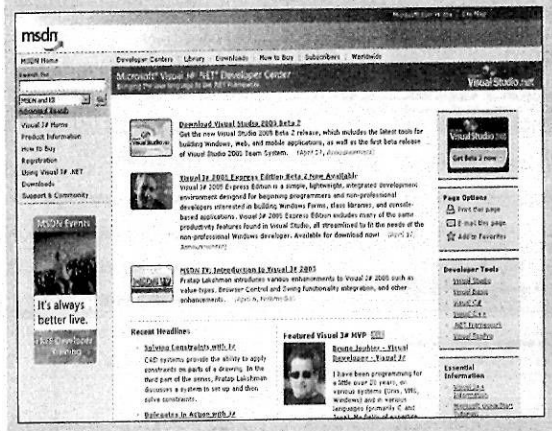
LENGUAJE BASE	COMPILADOR	ENLACE A SITIO OFICIAL
Ada	A#	www.usafa.af.mil/dfcs/bios/mcc_html/a_sharp.html
APL	Dyalog APL	www.dyalog.com/
Basic	Visual Basic .NET	http://msdn.microsoft.com/vbasic/
Basic	mbas	www.go-mono.com/mbas.html
C	lcc	www.cs.princeton.edu/software/lcc/
C#	C#	http://msdn.microsoft.com/vcsharp/
C#	mcs	www.go-mono.com/c-sharp.html



Con Mono podremos ejecutar nuestros programas creados en C# no sólo en Windows sino también en Linux, MacOsX y otras plataformas.

C++	Managed C++	http://msdn.microsoft.com/vstudio/techinfo/articles/upgrade/managedext.asp
Caml	F#	http://research.microsoft.com/projects/ilx/fsharp.aspx
Caml	OCAMIL	www.pps.jussieu.fr/~montela/ocamil/documentation.html
Cobol	NetCobol	www.netcobol.com/
Cobol	NetExpress	www.microfocus.com/products/netexpress/
Delphi	Delphi 2005	www.borland.com/delphi/
Delphi	Delphi.NET	http://sourceforge.net/projects/delphinet
Eiffel	Eiffel .NET	www.eiffel.com/
Forth	Delta Forth .NET	www.dataman.ro/dforth/

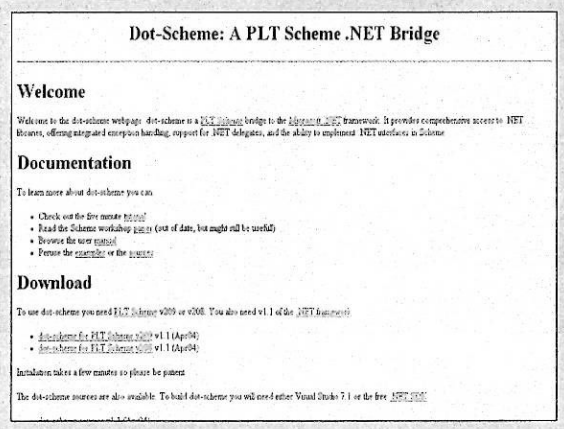
LENGUAJE BASE	COMPILADOR	ENLACE A SITIO OFICIAL
Fortran	FNT95	www.silverfrost.com/11/ftn95/ftn95_fortran_95_for_windows.asp
Haskell	Hugs98 for .NET	http://galois.com/~sof/hugs98.net/
Haskell	Mondrian	www.mondrian-script.org/
Haskell	Haskell.NET	http://php.cin.ufpe.br/~haskell/haskelldotnet/
Java	Visual J# .NET	http://msdn.microsoft.com/vjsharp/



J# es una implementación particular del lenguaje Java realizada por Microsoft. En vez de ejecutarse sobre una máquina virtual de Java, se ejecuta sobre la plataforma .NET.

Java	IKVM.NET	http://weblog.ikvm.net/
JavaScript	JScript .NET	www.gotdotnet.com/team/jscript/
JavaScript	JANET	http://janet-js.sourceforge.net/
Lexico	Lexico	http://riosur.net/
LISP	DotLisp	http://sourceforge.net/projects/dotlisp/
LISP	RDNZL	www.weitz.de/rdnzl/
LOGO	MonoLogo	http://monologo.sourceforge.net/
LOGO	TurtleTracks	www.mech.upatras.gr/~robgroup/logo/turtletracks/index.html
Lua	Lua.NET	www.lua.inf.puc-rio.br/luanet/
Mercury	Mercury on .NET	www.cs.mu.oz.au/research/mercury/dotnet.html
Mixal	Mixal	http://sourceforge.net/projects/mixnet/
Nemerle	Nemerle	http://nemerle.org/
Perl	PerlSharp	http://taubz.for.net/code/perlsharp/
Perl	PerlNET	http://aspn.activestate.com/ASPN/NET
Pascal	Chrome	www.chromesville.com/
Pascal	TMT .NET	www.tmt.com/net.htm
PHP	PHP Mono Extensions	www.php.net/~sterling/mono/
Prolog	P#	www.dcs.ed.ac.uk/home/jjc/psharp/psharp-1.1/dlpsharp.html
Python	boo	http://boo.codehaus.org/
Python	IronPython	http://ironpython.com/
Python	Python for .NET	http://starship.python.net/crew/mhammond/dotnet/

LENGUAJE BASE	COMPILADOR	ENLACE A SITIO OFICIAL
Ruby	NetRuby	www.geocities.co.jp/SiliconValley-PaloAlto/9251/ruby/nrb.html
Ruby	Ruby/.NET	www.saltypickle.com/rubydotnet/
Scala	Scala on .NET	http://scala.epfl.ch/index.html
Scheme	Larceny	www.ccs.neu.edu/home/will/Larceny/
Scheme	Dot-Scheme	www.rivendell.ws/dot-scheme/



Dot-Scheme: A PLT Scheme .NET Bridge

Welcome

Welcome to the dot-scheme webpage. dot-scheme is a PLT Scheme bridge to the Microsoft .NET framework. It provides comprehensive access to .NET libraries, offering integrated exception handling, support for .NET delegates, and the ability to implement .NET interfaces in Scheme.

Documentation

To learn more about dot-scheme you can:

- Check out the five minute tutorial
- Read the Scheme workshop paper (out of date, but might still be useful)
- Browse the user manual
- Peruse the examples or the source

Download

To use dot-scheme you need PLT Scheme v209 or v208. You also need v1.1 of the .NET framework.

- dot-scheme for PLT Scheme v209 v1.1 (Apr04)
- dot-scheme for PLT Scheme v208 v1.1 (Apr04)

Installation takes a few minutes so please be patient.

The dot-scheme sources are also available. To build dot-scheme you will need either Visual Studio 7.1 or the free .NET SDK.

Casi todos los lenguajes que al día de hoy mantienen cierta popularidad en la comunidad poseen sus versiones para .NET. Scheme no es una excepción.

Scheme	HotDog	http://rover.cs.northwestern.edu/~scheme/
Scheme	Tachy	www.kenrawlings.com/pages/Tachy
Scheme	Scheme.NET	www.cs.indiana.edu/~jgrinbla/index.htm
SmallTalk	#SmallTalk	www.refactory.com/Software/SharpSmallTalk/
SmallTalk	VMX SmallTalk	http://vmx-net.com/
SML	SML.NET	www.cl.cam.ac.uk/Research/TSG/SMLNET/
Spry	Spry	http://spry-lang.org/
Tcl/Tk	TickleSharp	http://forge.novell.com/modules/xfmod/project/?ticklesharp

ÍNDICE TEMÁTICO

EQUIVALENCIA DE TÉRMINOS

En este libro	Otras formas	En inglés
Acceso dial up	Acceso de marcación	
Actualización		Update, Upgrade
Actualizar		Refresh
Ancho de banda		Bandwidth
Archivos	Filas, Ficheros, Archivos electrónicos	Files
Archivos adjuntos	Archivos anexados o anexos	Attach, Attachment
Backup	Copia de respaldo, Copia de seguridad	
Balde de pintura	Bote de pintura	
Base de datos		Database
Booteo	Inicio/Arranque	Boot
Buscador		Search engine
Captura de pantalla		Snapshot
Carpeta		Folder
Casilla de correo	Buzón de correo	
CD-ROM	Disco compacto	Compact disk
Chequear	Checar, Verificar, Revisar	Check
Chip	Pastilla	
Cibercafé	Café de Internet	
Clipboard	Portapapeles	
Cliquear	Pinchar	
Colgar	Trabar	Tilt
Controlador	Adaptador	Driver
Correo electrónico		E-Mail, Electronic Mail, Mail
Descargar programas	Bajar programas, Telecargar programas	Download
Desfragmentar		Defrag
Destornillador	Desarmador	
Disco de inicio	Disco de arranque	Startup disk
Disco rígido	Disco duro, Disco fijo	Hard disk
Disquete	Disco flexible	Floppy drive
Firewall	Cortafuego	
Formatear		Format
Fuente		Font
Gabinete	Chasis, Cubierta	
Grabadora de CD	Quemadora de CD	CD Burn
Grupo de noticias		Newsgroup

En este libro	Otras formas	En inglés
Handheld	Computadora de mano	
Hipertexto		HyperText
Hospedaje de sitios	Alojamiento de sitios	Hosting
Hub	Concentrador	
Impresora		Printer
Inalámbrico		Wireless
Libro electrónico		E-Book
Lista de correo	Lista de distribución	Mailing list
Motherboard	Placa madre, Placa base	
Mouse	Ratón	
Navegador		Browser
Notebook	Computadora de mano, Computadora portátil	
Offline	Fuera de línea	
Online	En línea	
Página de inicio		Home page
Panel de control		Control panel
Parlantes	Bocinas, Altavoces	
PC	Computador, Ordenador, Computadora Personal, Equipo de cómputo	Personal Computer
Pestaña	Ficha, Solapa	
Pila	Batería	Battery
Placa de sonido		Soundboard
Plug & Play	Enchufar y usar	
Por defecto	Por predefinición	By default
Programas	Aplicación, Utilitarios	Software, Applications
Protector de pantalla		Screensaver
Proveedor de acceso a Internet		Internet Service Provider
Puente		Bridge
Puerto Serial		Serial Port
Ranura		Slot
Red		Net, Network
Servidor		Server
Sistema operativo	SO	Operating System (OS)
Sitio web	Site	
Tarjeta de video	Placa de video	
Tipear	Teclear, Escribir, Ingresar, Digitar	
Vinculo	Liga, Enlace, Hipervínculo, Hiperenlace	Link

ABREVIATURAS COMÚNMENTE UTILIZADAS

Abreviatura	Definición
ADSL	Asymmetric Digital Subscriber Line o Línea de abonado digital asimétrica
AGP	Accelerated Graphic Port o Puerto acelerado para gráficos
ANSI	American National Standards Institute
ASCII	American Standard Code of Information Interchange o Código americano estándar para el intercambio de información
BASIC	Beginner´s All-Purpose Symbolic Instruction Code
BIOS	Basic Input/Output System
Bit	Binary digit (Dígito binario)
Bps	Bits por segundo
CD	Compact Disk
CGI	Common Gateway Interface
CPU	Central Processing Unit o Unidad central de proceso
CRC	Cyclic Redundancy Checking
DNS	Domain Name System o Sistema de nombres de dominios
DPI	Dots per inch o puntos por pulgada
DVD	Digital Versatile Disc
FTP	File Transfer Protocol o Protocolo de transferencia de archivos
GB	Gigabyte
HTML	HyperText Mark-up Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Device Electronic
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IR	Infra Red
IRC	Internet Relay Chat
IRQ	Interrupt Request Line o Línea de petición de interrupción
ISO	International Organization Standard u Organización de Estándares Internacionales
ISP	Internet Service Provider o Proveedor de acceso a Internet
KB	Kilobyte
LAN	Local Area Network o Red de área local
LCD	Liquid Crystal Display o Pantalla de cristal líquido
LPT	Line Print Terminal
MB	Megabyte
MBR	Master Boot Record
MHz	Megahertz

Abreviatura	Definición
NETBEUI	Network Basic Extended User Interface o Interfaz de usuario extendida NETBios
OEM	Original Equipment Manufacturer
OS	Operative System
OSI	Open Systems Interconnection o Interconexión de sistemas abiertos
PCMCIA	Personal Computer Memory Card International Association
PDA	Personal Digital Assistant
PDF	Portable Document Format
Perl	Practical Extraction and Report Language
PGP	Pretty Good Privacy
PHP	Personal Home Page Tools, ahora llamado PHP Hypertext Preprocessor
POP3	Post Office Protocol 3 o versión 3 del Protocolo de oficina de correo
PPP	Point to Point Protocol o Protocolo punto a punto
RAM	Random Access Memory
ROM	Read Only Memory
SMTP	Simple Mail Transport Protocol o Protocolo simple de transferencia de correo
SPX/IPX	Sequence Packet eXchange/Internetwork Packet eXchange o Intercambio de paquetes secuenciales/Intercambio de paquetes entre redes
SQL	Structured Query Language
SSL	Secure Socket Layer
TCP/IP	Transfer Control Protocol / Internet Protocol o Protocolo de control de transferencia / Protocolo de Internet
UML	Lenguaje de Modelado Unificado
UDP	User Datagram Protocol
UPS	Uninterruptible Power Supply
URL	Uniform Resource Locator
USB	Universal Serial Bus
VGA	Video Graphic Array
WAN	Wide Area Network o Red de área extensa
WAP	Wireless Application Protocol
WWW	World Wide Web
XML	Extensible Markup Language