www.devjoker.com

Introduccion a C#

Origen y necesidad de un nuevo lenguaje

C# (leído en inglés “C Sharp” y en español “C Almohadilla”) es el nuevo lenguaje de propósito general diseñado por Microsoft para su plataforma .NET. Sus principales creadores son Scott Wiltamuth y Anders Hejlsberg, éste último también conocido por haber sido el diseñador del lenguaje Turbo Pascal y la herramienta RAD Delphi.

Aunque es posible escribir código para la plataforma .NET en muchos otros lenguajes, C# es el único que ha sido diseñado específicamente para ser utilizado en ella, por lo que programarla usando C# es mucho más sencillo e intuitivo que hacerlo con cualquiera de los otros lenguajes ya que C# carece de elementos heredados innecesarios en .NET. Por esta razón, se suele decir que C# es el **lenguaje nativo de .NET**

La sintaxis y estructuración de C# es muy parecida a la de C++ o Java, puesto que la intención de Microsoft es facilitar la migración de códigos escritos en estos lenguajes a C# y facilitar su aprendizaje a los desarrolladores habituados a ellos. Sin embargo, su sencillez y el alto nivel de productividad son comparables con los de Visual Basic.

Un lenguaje que hubiese sido ideal utilizar para estos menesteres es Java, pero debido a problemas con la empresa creadora del mismo -Sun-, Microsoft ha tenido que desarrollar un nuevo lenguaje que añadiese a las ya probadas virtudes de Java las modificaciones que Microsoft tenía pensado añadirle para mejorarlo aún más y hacerlo un lenguaje orientado al desarrollo de componentes.

En resumen, C# es un lenguaje de programación que toma las mejores características de lenguajes preexistentes como Visual Basic, Java o C++ y las combina en uno solo. El hecho de ser relativamente reciente no implica que sea inmaduro, pues Microsoft ha escrito la mayor parte de la BCL usándolo, por lo que su compilador es el más depurado y optimizado de los incluidos en el *.NET Framework SDK*

Características de C#

Con la idea de que los programadores más experimentados puedan obtener una visión general del lenguaje, a continuación se recoge de manera resumida las principales características de C# Alguna de las características aquí señaladas no son exactamente propias del lenguaje sino de la plataforma .NET en general, y si aquí se comentan es porque tienen una repercusión directa en el lenguaje:

- **Sencillez:** C# elimina muchos elementos que otros lenguajes incluyen y que son innecesarios en .NET. Por ejemplo:
  - El código escrito en C# es **autocontenido**, lo que significa que no necesita de ficheros adicionales al propio fuente tales como ficheros de cabecera o ficheros IDL
  - El tamaño de los tipos de datos básicos es fijo e independiente del compilador, sistema operativo o máquina para quienes se compile (no como en C++), lo que facilita la portabilidad del código.
  - No se incluyen elementos poco útiles de lenguajes como C++ tales como macros, herencia múltiple o la necesidad de un operador diferente del punto (**.**) acceder a miembros de espacios de nombres (**::**)
- **Modernidad:** C# incorpora en el propio lenguaje elementos que a lo largo de los años ha ido demostrándose son muy útiles para el desarrollo de aplicaciones y que en otros lenguajes como Java o C++ hay que simular, como un tipo básico **decimal** que permita realizar operaciones de alta precisión con reales de 128 bits (muy útil en el mundo financiero), la inclusión de una instrucción **foreach** que permita recorrer colecciones con facilidad y es ampliable a tipos definidos por el usuario, la inclusión de un tipo básico **string** para representar cadenas o la distinción de un tipo **bool** específico para representar valores lógicos.
- **Orientación a objetos:** Como todo lenguaje de programación de propósito general actual, C# es un lenguaje orientado a objetos, aunque eso es más bien una característica del CTS que de C#. Una diferencia de este enfoque orientado a objetos respecto al de otros lenguajes como C++ es que el de C# es más puro en tanto que no admiten ni funciones ni variables globales sino que todo el código y datos han de definirse dentro de definiciones de tipos de datos, lo que reduce problemas por conflictos de nombres y facilita la legibilidad del código.

  C# soporta todas las características propias del paradigma de programación orientada a objetos: **encapsulación**, **herencia** y **polimorfismo**.

  En lo referente a la encapsulación es importante señalar que aparte de los típicos modificadores **public**, **private** y **protected**, C# añade un cuarto modificador llamado **internal**, que puede combinarse con **protected** e indica que al elemento a cuya definición precede sólo puede accederse desde su mismo ensamblado.

  Respecto a la herencia -a diferencia de C++ y al igual que Java- C# sólo admite herencia simple de clases ya que la múltiple provoca más quebraderos de cabeza que facilidades y en la mayoría de los casos su utilidad puede ser simulada con facilidad mediante herencia múltiple de interfaces. De todos modos, esto vuelve a ser más bien una característica propia del CTS que de C#.

  Por otro lado y a diferencia de Java, en C# se ha optado por hacer que todos los métodos sean por defecto sellados y que los redefinibles hayan

de marcarse con el modificador **virtual** (como en C++), lo que permite evitar errores derivados de redefiniciones accidentales. Además, un efecto secundario de esto es que las llamadas a los métodos serán más eficientes por defecto al no tenerse que buscar en la tabla de funciones virtuales la implementación de los mismos a la que se ha de llamar. Otro efecto secundario es que permite que las llamadas a los métodos virtuales se puedan hacer más eficientemente al contribuir a que el tamaño de dicha tabla se reduzca.

- **Orientación a componentes:** La propia sintaxis de C# incluye elementos propios del diseño de componentes que otros lenguajes tienen que simular mediante construcciones más o menos complejas. Es decir, la sintaxis de C# permite definir cómodamente **propiedades** (similares a campos de acceso controlado), **eventos** (asociación controlada de funciones de respuesta a notificaciones) o **atributos** (información sobre un tipo o sus miembros)

- **Gestión automática de memoria:** Como ya se comentó, todo lenguaje de .NET tiene a su disposición el recolector de basura del CLR. Esto tiene el efecto en el lenguaje de que no es necesario incluir instrucciones de destrucción de objetos. Sin embargo, dado que la destrucción de los objetos a través del recolector de basura es indeterminista y sólo se realiza cuando éste se active –ya sea por falta de memoria, finalización de la aplicación o solicitud explícita en el fuente-, C# también proporciona un mecanismo de liberación de recursos determinista a través de la instrucción **using**.

- **Seguridad de tipos:** C# incluye mecanismos que permiten asegurar que los accesos a tipos de datos siempre se realicen correctamente, lo que permite evita que se produzcan errores difíciles de detectar por acceso a memoria no perteneciente a ningún objeto y es especialmente necesario en un entorno gestionado por un recolector de basura. Para ello se toman medidas del tipo:

    - Sólo se admiten **conversiones entre tipos compatibles**. Esto es, entre un tipo y antecesores suyos, entre tipos para los que explícitamente se haya definido un operador de conversión, y entre un tipo y un tipo hijo suyo del que un objeto del primero almacenase una referencia del segundo (**downcasting**) Obviamente, lo último sólo puede comprobarlo en tiempo de ejecución el CLR y no el compilador, por lo que en realidad el CLR y el compilador colaboran para asegurar la corrección de las conversiones.

    - No se pueden usar **variables no inicializadas**. El compilador da a los campos un valor por defecto consistente en ponerlos a cero y controla mediante análisis del flujo de control del fuente que no se lea ninguna variable local sin que se le haya asignado previamente algún valor.

    - Se comprueba que todo **acceso a los elementos de una tabla** se realice con índices que se encuentren dentro del rango de la misma.

    - Se puede controlar la **producción de desbordamientos** en operaciones aritméticas, informándose de ello con una excepción cuando ocurra. Sin embargo, para conseguirse un mayor rendimiento en la aritmética estas comprobaciones no se hacen por defecto al operar con variables sino sólo con constantes (se pueden detectar en tiempo de compilación)

    - A diferencia de Java, C# incluye **delegados**, que son similares a los punteros a funciones de C++ pero siguen un enfoque orientado a objetos, pueden almacenar referencias a varios métodos simultáneamente, y se comprueba que los métodos a los que apunten tengan parámetros y valor de retorno del tipo indicado al definirlos.

    - Pueden definirse métodos que admitan un número indefinido de parámetros de un cierto tipo, y a diferencia lenguajes como C/C++, en C# siempre se comprueba que los valores que se les pasen en cada llamada sean de los tipos apropiados.

- **Instrucciones seguras:** Para evitar errores muy comunes, en C# se han impuesto una serie de restricciones en el uso de las instrucciones de control más comunes. Por ejemplo, la guarda de toda condición ha de ser una expresión condicional y no aritmética, con lo que se evitan errores por confusión del operador de igualdad (==) con el de asignación (=); y todo caso de un **switch** ha de terminar en un **break** o **goto** que indique cuál es la siguiente acción a realizar, lo que evita la ejecución accidental de casos y facilita su reordenación.

- **Sistema de tipos unificado:** A diferencia de C++, en C# todos los tipos de datos que se definan siempre derivarán, aunque sea de manera implícita, de una clase base común llamada **System.Object**, por lo que dispondrán de todos los miembros definidos en ésta clase (es decir, serán "objetos")

  A diferencia de Java, en C# esto también es aplicable a los tipos de datos básicos Además, para conseguir que ello no tenga una repercusión negativa en su nivel de rendimiento, se ha incluido un mecanismo transparente de **boxing** y **unboxing** con el que se consigue que sólo sean tratados como objetos cuando la situación lo requiera, y mientras tanto puede aplicárseles optimizaciones específicas.

  El hecho de que todos los tipos del lenguaje deriven de una clase común facilita enormemente el diseño de colecciones genéricas que puedan almacenar objetos de cualquier tipo.

- **Extensibilidad de tipos básicos:** C# permite definir, a través de **estructuras**, tipos de datos para los que se apliquen las mismas optimizaciones que para los tipos de datos básicos. Es decir, que se puedan almacenar directamente en pila (luego su creación, destrucción y acceso serán más rápidos) y se asignen por valor y no por referencia. Para conseguir que lo último no tenga efectos negativos al pasar estructuras como parámetros de métodos, se da la posibilidad de pasar referencias a pila a través del modificador de parámetro **ref**.

- **Extensibilidad de operadores:** Para facilitar la legibilidad del código y conseguir que los nuevos tipos de datos básicos que se definan a través de las estructuras estén al mismo nivel que los básicos predefinidos en el lenguaje, al igual que C++ y a diferencia de Java, C# permite redefinir el significado de la mayoría de los operadores -incluidos los de conversión, tanto para conversiones implícitas como explícitas- cuando se apliquen a diferentes tipos de objetos.

  Las redefiniciones de operadores se hacen de manera inteligente, de modo que a partir de una única definición de los operadores ++ y -- el compilador puede deducir automáticamente como ejecutarlos de manera prefijas y postifja; y definiendo operadores simples (como +), el compilador deduce cómo aplicar su versión de asignación compuesta (+=) Además, para asegurar la consistencia, el compilador vigila que los operadores con opuesto siempre se redefinan por parejas (por ejemplo, si se redefine ==, también hay que redefinir **!=**)

  También se da la posibilidad, a través del concepto de **indizador**, de redefinir el significado del operador **[]** para los tipos de dato definidos por el usuario, con lo que se consigue que se pueda acceder al mismo como si fuese una tabla. Esto es muy útil para trabajar con tipos que actúen como

colecciones de objetos.

- **Extensibilidad de modificadores:** C# ofrece, a través del concepto de **atributos**, la posibilidad de añadir a los metadatos del módulo resultante de la compilación de cualquier fuente información adicional a la generada por el compilador que luego podrá ser consultada en tiempo ejecución a través de la librería de reflexión de .NET . Esto, que más bien es una característica propia de la plataforma .NET y no de C#, puede usarse como un mecanismo para definir nuevos modificadores.

- **Versionable:** C# incluye una **política de versionado** que permite crear nuevas versiones de tipos sin temor a que la introducción de nuevos miembros provoquen errores difíciles de detectar en tipos hijos previamente desarrollados y ya extendidos con miembros de igual nombre a los recién introducidos.

  Si una clase introduce un nuevo método cuyas redefiniciones deban seguir la regla de llamar a la versión de su padre en algún punto de su código, difícilmente seguirían esta regla miembros de su misma signatura definidos en clases hijas previamente a la definición del mismo en la clase padre; o si introduce un nuevo campo con el mismo nombre que algún método de una clase hija, la clase hija dejará de funcionar. Para evitar que esto ocurra, en C# se toman dos medidas:

  - Se obliga a que toda redefinición deba incluir el modificador **override**, con lo que la versión de la clase hija nunca sería considerada como una redefinición de la versión de miembro en la clase padre ya que no incluiría **override**. Para evitar que por accidente un programador incluya este modificador, sólo se permite incluirlo en miembros que tengan la misma signatura que miembros marcados como redefinibles mediante el modificador **virtual**. Así además se evita el error tan frecuente en Java de creerse haber redefinido un miembro, pues si el miembro con **override** no existe en la clase padre se producirá un error de compilación.

  - Si no se considera redefinición, entonces se considera que lo que se desea es ocultar el método de la clase padre, de modo que para la clase hija sea como si nunca hubiese existido. El compilador avisará de esta decisión a través de un mensaje de aviso que puede suprimirse incluyendo el modificador **new** en la definición del miembro en la clase hija para así indicarle explícitamente la intención de ocultación.

- **Eficiente:** En principio, en C# todo el código incluye numerosas restricciones para asegurar su seguridad y no permite el uso de punteros. Sin embargo, y a diferencia de Java, en C# es posible saltarse dichas restricciones manipulando objetos a través de punteros. Para ello basta marcar regiones de código como inseguras (modificador **unsafe**) y podrán usarse en ellas punteros de forma similar a cómo se hace en C++, lo que puede resultar vital para situaciones donde se necesite una eficiencia y velocidad procesamiento muy grandes.

- **Compatible:** Para facilitar la migración de programadores, C# no sólo mantiene una sintaxis muy similar a C, C++ o Java que permite incluir directamente en código escrito en C# fragmentos de código escrito en estos lenguajes, sino que el CLR también ofrece, a través de los llamados **Platform Invocation Services** (**PInvoke**), la posibilidad de acceder a código nativo escrito como funciones sueltas no orientadas a objetos tales como las DLLs de la API Win32. Nótese que la capacidad de usar punteros en código inseguro permite que se pueda acceder con facilidad a este tipo de funciones, ya que éstas muchas veces esperan recibir o devuelven punteros.

  También es posible acceder desde código escrito en C# a objetos COM. Para facilitar esto, el *.NET Framework SDK* incluye una herramientas llamadas **tlbimp** y **regasm** mediante las que es posible generar automáticamente clases proxy que permitan, respectivamente, usar objetos COM desde .NET como si de objetos .NET se tratase y registrar objetos .NET para su uso desde COM.

  Finalmente, también se da la posibilidad de usar controles ActiveX desde código .NET y viceversa. Para lo primero se utiliza la utilidad **aximp**, mientras que para lo segundo se usa la ya mencionada **regasm**.

www.devjoker.com

Programacion con C#

Aplicación básica ¡Hola Mundo!

Básicamente una aplicación en C# puede verse como un conjunto de uno o más ficheros de código fuente con las instrucciones necesarias para que la aplicación funcione como se desea y que son pasados al compilador para que genere un ejecutable. Cada uno de estos ficheros no es más que un fichero de texto plano escrito usando caracteres Unicode y siguiendo la sintaxis propia de C#.

Como primer contacto con el lenguaje, nada mejor que el típico programa de iniciación "¡Hola Mundo!" que lo único que hace al ejecutarse es mostrar por pantalla el mensaje ¡Hola Mundo! Su código es:[2]

```
1:          class HolaMundo
2:          {
3:                      static void Main()
4:                      {
5:                              System.Console.WriteLine("¡Hola Mundo!");
6:                      }
7:          }
```

Todo el código escrito en C# se ha de escribir dentro de una definición de clase, y lo que en la línea **1:** se dice es que se va a definir una clase (**class**) de nombre HolaMundo1 cuya definición estará comprendida entre la llave de apertura de la línea **2:** y su correspondiente llave de cierre en la línea línea **7:**

Dentro de la definición de la clase (línea **3:**) se define un método de nombre Main cuyo código es el indicado entre la llave de apertura de la línea **4:** y su respectiva llave de cierre (línea **6:**) Un método no es más que un conjunto de instrucciones a las que se les asocia un nombre, de modo que para posteriormente ejecutarlas baste referenciarlas por su nombre en vez de tener que rescribirlas.

La partícula que antecede al nombre del método indica cuál es el tipo de valor que se devuelve tras la ejecución del método, y en este caso es **void** que significa que no se devuelve nada. Por su parte, los paréntesis colocados tras el nombre del método indican cuáles son los parámetros que éste toma, y el que estén vacíos significa que el método no toma ninguno. Los parámetros de un método permiten modificar el resultado de su ejecución en función de los valores que se les dé en cada llamada.

La palabra **static** que antecede a la declaración del tipo de valor devuelto es un **modificador** del significado de la declaración de método que indica que el método está asociado a la clase dentro de la que se define y no a los objetos que se creen a partir de ella. Main() es lo que se denomina el **punto de entrada** de la aplicación, que no es más que el método por el que comenzará su ejecución. Necesita del modificador **static** para evitar que para llamarlo haya que crear algún objeto de la clase donde se haya definido.

Finalmente, la línea **5:** contiene la instrucción con el código a ejecutar, que lo que se hace es solicitar la ejecución del método **WriteLine()** de la clase **Console** definida en el espacio de nombres **System** pasándole como parámetro la cadena de texto con el contenido ¡Hola Mundo! Nótese que las cadenas de textos son secuencias de caracteres delimitadas por comillas dobles aunque dichas comillas no forman parte de la cadena. Por su parte, un espacio de nombres puede considerarse que es para las clases algo similar a lo que un directorio es para los ficheros: una forma de agruparlas.

El método **WriteLine()** se usará muy a menudo en los próximos temas, por lo que es conveniente señalar ahora que una forma de llamarlo que se utilizará en repetidas ocasiones consiste en pasarle un número indefinido de otros parámetros de cualquier tipo e incluir en el primero subcadenas de la forma {i}. Con ello se consigue que se muestre por la ventana de consola la cadena que se le pasa como primer parámetro pero sustituyéndole las subcadenas {i} por el valor convertido en cadena de texto del parámetro que ocupe la posición i+2 en la llamada a **WriteLine()**. Por ejemplo, la siguiente instrucción mostraría Tengo 5 años por pantalla si x valiese 5:

```
System.Console.WriteLine("Tengo {0} años", x);
```

Para indicar cómo convertir cada objeto en un cadena de texto basta redefinir su método **ToString()**, aunque esto es algo que no se verá hasta el *Tema 5: Clases*.

Antes de seguir es importante resaltar que C# es sensible a las mayúsculas, los que significa que no da igual la capitalización con la que se escriban los identificadores. Es decir, no es lo mismo escribir Console que COnsole o CONSOLE, y si se hace de alguna de las dos últimas formas el compilador producirá un error debido a que en el espacio de nombres **System** no existe ninguna clase con dichos nombres. En este sentido, cabe señalar que un error común entre programadores acostumbrados a Java es llamar al punto de entrada main en vez de Main, lo que provoca un error al compilar ejecutables en tanto que el compilador no detectará ninguna definición de punto de entrada.

Puntos de entrada

Ya se ha dicho que el **punto de entrada** de una aplicación es un método de nombre Main que contendrá el código por donde se ha de iniciar la ejecución de la misma. Hasta ahora sólo se ha visto una versión de Main() que no toma parámetros y tiene como tipo de retorno **void**, pero en realidad todas sus posibles versiones son:

```
static void Main()

static int Main()

static int Main(string[] args)

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

Como se ve, hay versiones de Main() que devuelven un valor de tipo **int**. Un **int** no es más que un tipo de datos capaz de almacenar valor enteros comprendidos entre –2.147$_1$483.648 y 2.147$_1$483.647, y el número devuelto por Main() sería interpretado como código de retorno de la aplicación. Éste valor suele usarse para indicar si la aplicación a terminado con éxito (generalmente valor 0) o no (valor según la causa de la terminación anormal), y en el *Tema 8: Métodos* se explicará como devolver valores.

También hay versiones de Main() que toman un parámetro donde se almacenará la lista de argumentos con los que se llamó a la aplicación, por lo que sólo es útil usar estas versiones del punto de entrada si la aplicación va a utilizar dichos argumentos para algo. El tipo de este parámetro es **string[]**, lo que significa que es una tabla de cadenas de texto (en el *Tema 5: Claes* se explicará detenidamente qué son las tablas y las cadenas), y su nombre -que es el que habrá de usarse dentro del código de Main() para hacerle referencia- es args en el ejemplo, aunque podría dársele cualquier otro

Compilación en línea de comandos

Una vez escrito el código anterior con algún editor de textos –como el **Bloc de Notas** de Windows- y almacenado en formato de texto plano en un fichero HolaMundo.cs[3], para compilarlo basta abrir una ventana de consola (MS-DOS en Windows), colocarse en el directorio donde se encuentre y pasárselo como parámetro al compilador así:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc HolaMundo.cs
```

csc.exe es el compilador de C# incluido en el .NET Framework SDK para Windows de Microsoft. Aunque en principio el programa de instalación del SDK lo añade automáticamente al path para poder llamarlo sin problemas desde cualquier directorio, si lo ha instalado a través de VS.NET esto no ocurrirá y deberá configurárselo ya sea manualmente, o bien ejecutando el fichero por lotes **Common7\Tools\vsvars32.bat** que VS.NET incluye bajo su directorio de instalación, o abriendo la ventana de consola desde el icono **Herramientas de Visual Studio.NET à Símbolo del sistema de Visual Studio.NET** correspondiente al grupo de programas de VS.NET en el menú Inicio de Windows que no hace más que abrir la ventana de consola y llamar automáticamente a **vsvars32.bat**. En cualquier caso, si usa otros compiladores de C# puede que varie la forma de realizar la compilación, por lo que lo que aquí se explica en principio sólo será válido para los compiladores de C# de Microsoft para Windows.

Tras la compilación se obtendría un ejecutable llamado HolaMundo.exe cuya ejecución produciría la siguiente salida por la ventana de consola:

```
Hola Mundo!
```

Si la aplicación que se vaya a compilar no utilizase la ventana de consola para mostrar su salida sino una interfaz gráfica de ventanas, entonces habría que compilarla pasando al compilador la opción **/t** con el valor **winexe** antes del nombre del fichero a compilar. Si no se hiciese así se abriría la ventana de consola cada vez que ejecutase la aplicación de ventanas, lo que suele ser indeseable en este tipo de aplicaciones. Así, para compilar Ventanas.cs como ejecutable de ventanas sería conveniente escribir:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc /t:winexe Ventanas.cs
```

Nótese que aunque el nombre **winexe** dé la sensación de que este valor para la opción **/t** sólo permite generar ejecutables de ventanas, en realidad lo que permite es generar ejecutables sin ventana de consola asociada. Por tanto, también puede usarse para generar ejecutables que no tengan ninguna interfaz asociada, ni de consola ni gráfica.

Si en lugar de un ejecutable -ya sea de consola o de ventanas- se desea obtener una librería, entonces al compilar hay que pasar al compilador la opción **/t** con el valor **library**. Por ejemplo, siguiendo con el ejemplo inicial habría que escribir:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc /t:library HolaMundo.cs
```

En este caso se generaría un fichero HolaMundo.dll cuyos tipos de datos podrían utilizarse desde otros fuentes pasando al compilador una referencia a los mismos mediante la opción **/r**. Por ejemplo, para compilar como ejecutable un fuente A.cs que use la clase HolaMundo de la librería HolaMundo.dll se escribiría:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc /r:HolaMundo.dll A.cs
```

En general **/r** permite referenciar a tipos definidos en cualquier ensamblado, por lo que el valor que se le indique también puede ser el nombre de un ejecutable. Además, en cada compilación es posible referenciar múltiples ensamblados ya sea incluyendo la opción **/r** una vez por cada uno o incluyendo múltiples referencias en una única opción **/r** usando comas o puntos y comas como separadores. Por ejemplo, las siguientes tres llamadas al compilador son equivalentes:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc
/r:HolaMundo.dll;Otro.dll;OtroMás.exe  A.cs

C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc
/r:HolaMundo.dll,Otro.dll,OtroMás.exe A.cs

C:\WINDOWS\....\v2.0.50727>csc /t:HolaMundo.dll /r:Otro.dll /r:OtroMás.exe A.cs
```

Hay que señalar que aunque no se indique nada, en toda compilación siempre se referencia por defecto a la librería **mscorlib.dll** de la BCL, que incluye los tipos de uso más frecuente. Si se usan tipos de la BCL no incluidos en ella habrá que incluir al compilar referencias a las librerías donde estén definidos (en la documentación del SDK sobre cada tipo de la BCL puede encontrar información sobre donde se definió)

Tanto las librerías como los ejecutables son ensamblados. Para generar un módulo de código que no forme parte de ningún ensamblado sino que contenga definiciones de tipos que puedan añadirse a ensamblados que se compilen posteriormente, el valor que ha de darse al compilar a la opción **/t** es **module**. Por ejemplo:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc /t:module HolaMundo.cs
```

Con la instrucción anterior se generaría un módulo llamado HolaMundo.netmodule que podría ser añadido a compilaciones de ensamblados incluyéndolo como valor de la opción **/addmodule**. Por ejemplo, para añadir el módulo anterior a la compilación del fuente librería Lib.cs como librería se escribiría:

```
C:\WINDOWS\...\v2.0.50727>csc /t:library /addmodule:HolaMundo.netmodule Lib.cs
```

Aunque hasta ahora todas las compilaciones de ejemplo se han realizado utilizando un único fichero de código fuente, en realidad nada impide que se puedan utilizar más. Por ejemplo, para compilar los ficheros A.cs y B.cs en una librería A.dll se ejecutaría:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc /t:library A.cs B.cs
```

Nótese que el nombre que por defecto se dé al ejecutable generado siempre es igual al del primer fuente especificado pero con la extensión propia del tipo de compilación realizada (**.exe** para ejecutables, **.dll** para librerías y **.netmodule** para módulos) Sin embargo, puede especificárse como valor en la opción **/out** del compilador cualquier otro tal y como muestra el siguiente ejemplo que compila el fichero A.cs como una librería de nombre Lib.exe:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc /t:library /out:Lib.exe A.cs
```

Véase que aunque se haya dado un nombre terminado en **.exe** al fichero resultante, éste sigue siendo una librería y no un ejecutable e intentar ejecutarlo produciría un mensaje de error. Obviamente no tiene mucho sentido darle esa extensión, y sólo se le ha dado en este ejemplo para demostrar que, aunque recomendable, la extensión del fichero no tiene porqué corresponderse realmente con el tipo de fichero del que se trate.

A la hora de especificar ficheros a compilar también se pueden utilizar los caracteres de comodín típicos del sistema operativo. Por ejemplo, para compilar todos los ficheros con extensión .cs del directorio actual en una librería llamada Varios.dll se haría:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc /t:library /out:varios.dll *.cs
```

Con lo que hay que tener cuidado, y en especial al compilar varios fuentes, es con que no se compilen a la vez más de un tipo de dato con punto de entrada, pues entonces el compilador no sabría cuál usar como inicio de la aplicación. Para orientarlo, puede especificarse como valor de la opción **/main** el nombre del tipo que contenga el Main() ha usar como punto de entrada. Así, para compilar los ficheros A.cs y B.cs en un ejecutable cuyo punto de entrada sea el definido en el tipo Principal, habría que escribir:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc /main:Principal A.cs B.cs
```

Lógicamente, para que esto funcione A.cs o B.cs tiene que contener alguna definición de algún tipo llamado Principal con un único método válido como punto de entrada (obviamente, si contiene varios se volvería a tener el problema de no saber cuál utilizar)

Compilación con Visual Studio.NET

Para compilar una aplicación en Visual Studio.NET primero hay que incluirla dentro de algún proyecto. Para ello basta pulsar el botón **New Project** en la página de inicio que se muestra nada más arrancar dicha herramienta, tras lo que se obtendrá una pantalla con el aspecto mostrado en la **Ilustración 1** **08D0C9EA79F9BACE118C8200AA004BA90B02000000080000000E0000005F005200650066003400380034003600390037003900330033000000000** .

En el recuadro de la ventana mostrada etiquetado como **Project Types** se ha de seleccionar el tipo de proyecto a crear. Obviamente, si se va a trabajar en C# la opción que habrá que escoger en la misma será siempre Visual C# Projects.

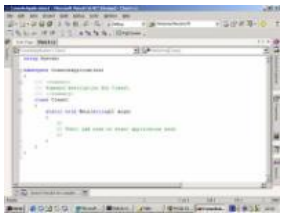
En el recuadro **Templates** se ha de seleccionar la plantilla correspondiente al subtipo de proyecto dentro del tipo indicado en **Project Types** que se va a realizar. Para realizar un ejecutable de consola, como es nuestro caso, hay que seleccionar el icono etiquetado como Console Application. Si se quisiese realizar una librería habría que seleccionar Class Library, y si se quisies realizar un ejecutable de ventanas habría que seleccionar Windows Application. Nótese que no se ofrece ninguna plantilla para realizar módulos, lo que se debe a que desde Visual Studio.NET no pueden crearse.

Por último, en el recuadro de texto **Name** se ha de escribir el nombre a dar al proyecto y en **Location** el del directorio base asociado al mismo. Nótese que bajo de **Location** aparecerá un mensaje informando sobre cual será el directorio donde finalmente se almacenarán los archivos del proyecto, que será el resultante de concatenar la ruta especificada para el directorio base y el nombre del proyecto.

[Ampliar Imagen]

Ilustración 1 : Ventana de creación de nuevo proyecto en Visual Studio.NET

Una vez configuradas todas estas opciones, al pulsar botón **OK** Visual Studio creará toda la infraestructura adecuada para empezar a trabajar cómodamente en el proyecto. Como puede apreciarse en la **Ilustración 2** , esta infraestructura consistirá en la generación de un fuente que servirá de plantilla para la realización de proyectos del tipo elegido (en nuestro caso, aplicaciones de consola en C#):



[Ampliar Imagen]

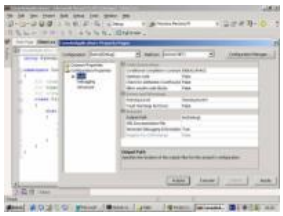
Ilustración 2 : Plantilla para aplicaciones de consola generada por Visual Studio.NET

A partir de esta plantilla, escribir el código de la aplicación de ejemplo es tan sencillo con simplemente teclear System.Console.WriteLine("¡Hola Mundo!") dentro de la definición del método Main() creada por Visual Studio.NET. Claro está, otra posibilidad es borrar toda la plantilla y sustituirla por el código para HolaMundo mostrado anteriormente.

Sea haga como se haga, para compilar y ejecutar tras ello la aplicación sólo hay que pulsar **CTRL+F5** o seleccionar **Debug à Start Without Debugging** en el menú principal de Visual Studio.NET. Para sólo compilar el proyecto, entonces hay que seleccionar **Build à Rebuild All**. De todas formas, en ambos casos el ejecutable generado se almacenará en el subdirectorio Bin\Debug del directorio del proyecto.

En el extremo derecho de la ventana principal de Visual Studio.NET puede encontrar el denominado **Solution Explorer** (si no lo encuentra, seleccione **View à Solution Explorer**), que es una herramienta que permite consultar cuáles son los archivos que forman el proyecto. Si selecciona en él el icono correspondiente al proyecto en que estamos trabajando y pulsa **View à Property Pages** obtendrá una hoja de propiedades del proyecto con el aspecto mostrado

en la **Ilustración 3** :



[Ampliar Imagen]

Ilustración 3 : Hoja de propiedades del proyecto en Visual Studio.NET

Esta ventana permite configurar de manera visual la mayoría de opciones con las que se llamará al compilador en línea de comandos. Por ejemplo, para cambiar el nombre del fichero de salida (opción **/out**) se indica su nuevo nombre en el cuadro de texto **Common Properties à General à Assembly Name**, para cambiar el tipo de proyecto a generar (opción **/t**) se utiliza **Common Properties à General à Output Type** (como verá si intenta cambiarlo, no es posible generar módulos desde Visual Studio.NET), y el tipo que contiene el punto de entrada a utilizar (opción **/main**) se indica en **Common Properties à General à Startup Object**

Finalemente, para añadir al proyecto referencias a ensamblados externos (opción **/r**) basta seleccionar **Project à Add Reference** en el menú principal de VS.NET.

www.devjoker.com

Fundamentos de C#

Comentarios

Un **comentario** es texto que se incluye en el código fuente para facilitarsu lectura a los programadores y cuyo contenido es, por defecto, completamenteignorado por el compilador. Suelen usarse para incluir información sobre elautor del código, para aclarar el significado o el porqué de determinadassecciones de código, para describir el funcionamiento de los métodos de lasclases, etc.

En C# hay dos formas de escribir comentarios. La primera consiste en encerrar todo el texto que se desee comentar entre caracteres **/*** y ***/** . Estos comentarios pueden abarcar tantas líneas como sea necesario. Porejemplo:

```
/* Texto */

/* Esto es un comentario que ejemplifica cómo se escribe
comentarios que ocupen varias líneas*/
```

Ahora bien, hay quetener cuidado con el hecho de que no es posible anidar comentarios de este tipo. Es decir, no vale escribir comentarios como el siguiente:

```
NO ES CORRECTO!
/* Comentario contenedor /* Comentario contenido */  */
```

Esto se debe a que como el compilador ignora todo el texto contenido en un comentario y sólo busca la secuencia */ que marca su final, ignorará el segundo /* y cuando llegue al primer */ considerará que ha acabado el comentario abierto con el primer /* (no el abierto con el segundo) y pasará a buscar código. Como el */ sólo lo admite si ha detectado antes algún comentario abierto y aún no cerrado (no mientras busca código), cuando llegue al segundo */ considerará que ha habido un error ya que encontrará el */ donde esperaba encontrar código

Dado que muchas veces los comentarios que se escriben son muy cortos y no suelen ocupar más de una línea, C# ofrece una sintaxis alternativa más compacta para la escritura este tipo de comentarios en las que se considera como indicador del comienzo del comentario la pareja de caracteres // y como indicador de su final el fin de línea. Por tanto, la sintaxis que siguen estos comentarios es:

```
// <texto>
```

Y un ejemplo de su uso es:

```
// Este comentario ejemplifica como escribir comentarios de una sola línea
```

Estos comentarios de una sola línea sí que pueden anidarse sin ningún problema. Por ejemplo, el siguiente comentario es perfectamente válido:

```
// Comentario contenedor // Comentario contenido
```

Identificadores

Al igual que en cualquier lenguaje de programación, en C# un identificador no es más que, como su propio nombre indica, un nombre con el que identificaremos algún elemento de nuestro código, ya sea una clase, una variable, un método, etc.

Típicamente el nombre de un identificador será una secuencia de cualquier número de caracteres alfanuméricos –incluidas vocales acentuadas y eñes- tales que el primero de ellos no sea un número. Por ejemplo, identificadores válidos serían: Arriba, caña, C3P0, áëÎò, etc; pero no lo serían 3com, 127, etc.

Sin embargo, y aunque por motivos de legibilidad del código no se recomienda, C# también permite incluir dentro de un identificador caracteres especiales imprimibles tales como símbolos de diéresis, subrayados, etc. siempre y cuando estos no tengan un significado especial dentro del lenguaje. Por ejemplo, también serían identificadores válidos, _barco_, c¨k y A·B; pero no C# (# indica inicio de directiva de preprocesado) o a!b (! indica operación lógica “not”)

Finalmente, C# da la posibilidad de poder escribir identificadores que incluyan caracteres Unicode que no se puedan imprimir usando el teclado de la máquina del programador o que no sean directamente válidos debido a que tengan un significado especial en el lenguaje. Para ello, lo que permite es escribir estos caracteres usando secuencias de escape, que no son más que secuencias de caracteres con las sintaxis:

\u<dígito><dígito><dígito><dígito>

ó \U<dígito><dígito><dígito><dígito><dígito><dígito><dígito><dígito>

Estos dígitos indican es el código Unicode del carácter que se desea incluir como parte del identificador, y cada uno de ellos ha de ser un dígito hexadecimal válido. (0-9, a-f ó A-F) Hay que señalar que el carácter u ha de escribise en minúscula cuando se indiquen caracteres Unicode con 4 dígitos y en mayúscula cuando se indiquen con caracteres de ocho. Ejemplos de identificadores válidos son C\u0064 (equivale a C#, pues 64 es el código de # en Unicode) ó a\U00000033b (equivale a a!b)

Palabras reservadas

Aunque antes se han dado una serie de restricciones sobre cuáles son los nombres válidos que se pueden dar en C# a los identificadores, falta todavía por dar una: los siguientes nombres no son válidos como identificadores ya que tienen un significado especial en el lenguaje:

```
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
```

Aparte de estas palabras reservadas, si en futuras implementaciones del lenguaje se decidiese incluir nuevas palabras reservadas, Microsoft dice que dichas palabras habrían de incluir al menos dos símbolos de subrayado consecutivos (__) Por tanto, para evitar posibles conflictos futuros no se recomienda dar a nuestros identificadores nombres que contengan dicha secuencia de símbolos.

Aunque directamente no podemos dar estos nombres a nuestros identificadores, C# proporciona un mecanismo para hacerlo indirectamente y de una forma mucho más legible que usando secuencias de escape. Este mecanismo consiste en usar el carácter @ para prefijar el nombre coincidente con el de una palabra reservada que queramos dar a nuestra variable. Por ejemplo, el siguiente código es válido:

```
class @class
   {
      static void @static(bool @bool)
      {
         if (@bool)
            Console.WriteLine("cierto");
         else
            Console.WriteLine("falso");
      }
   }
```

Lo que se ha hecho en el código anterior ha sido usar @ para declarar una clase de nombre class con un método de nombre static que toma un parámetro de nombre bool, aún cuando todos estos nombres son palabras reservadas en C#.

Hay que precisar que aunque el nombre que nosotros escribamos sea por ejemplo @class, el nombre con el que el compilador va a tratar internamente al identificador es solamente class. De hecho, si desde código escrito en otro lenguaje adaptado a .NET distinto a C# hacemos referencia a éste identificador y en ese lenguaje su nombre no es una palabra reservada, el nombre con el que deberemos referenciarlo es class, y no @class (si también fuese en ese lenguaje palabra reservada habría que referenciarlo con el mecanismo que el lenguaje incluyese para ello, que quizás también podría consistir en usar @ como en C#)

En realidad, el uso de @ no se tiene porqué limitar a preceder palabras reservadas en C#, sino que podemos preceder cualquier nombre con él. Sin embargo, hacer esto no se recomienda, pues es considerado como un mal hábito de programación y puede provocar errores muy sutiles como el que muestra el siguiente ejemplo:

```
class A
{
```

```
    int a;      // (1)
    int @a;     // (2)
    public static void Main()
    {}
}
```

Si intentamos compilar este código se producirá un error que nos informará de que el campo de nombre a ha sido declarado múltiples veces en la clase A. Esto se debe a que como @ no forma parte en realidad del nombre del identificador al que precede, las declaraciones marcadas con comentarios como (1) y (2) son equivalentes.

Hay que señalar por último una cosa respecto al carácter @: sólo puede preceder al nombre de un identificador, pero no puede estar contenido dentro del mismo. Es decir, identificadores como i5322@fie.us.es no son válidos.

Literales

Un literal es la representación explícita de los valores que pueden tomar los tipos básicos del lenguaje. A continuación se explica cuál es la sintaxis con que se escriben los literales en C# desglosándolos según el tipo de valores que representan:

- **Literales enteros: Un número entero se puede representar en C# tanto en formato decimal como hexadecimal. En el primer caso basta escribir los dígitos decimales (0-9) del número unos tras otros, mientras que en el segundo hay que preceder los dígitos hexadecimales (0-9, a-f, A-F) con el prefijo 0x. En ambos casos es posible preceder el número de los operadores + ó – para indicar si es positivo o negativo, aunque si no se pone nada se considerará que es positivo. Ejemplos de literales enteros son 0, 5, +15, -23, 0x1A, -0x1a, etc**

En realidad, la sintaxis completa para la escritura de literales enteros también puede incluir un sufijo que indique el tipo de dato entero al que ha de pertenecer el literal. Esto no lo veremos hasta el *Tema 7: Variables y tipos de datos*.

- **Literales reales: Los números reales se escriben de forma similar a los enteros, aunque sólo se pueden escribir en forma decimal y para separar la parte entera de la real usan el tradicional punto decimal (carácter .) También es posible representar los reales en formato científico, usándose para indicar el exponente los caracteres e ó E. Ejemplos de literales reales son 0.0, 5.1, -5.1, +15.21, 3.02e10, 2.02e-2, 98.8E+1, etc.**

Al igual que ocurría con los literales enteros, los literales reales también pueden incluir sufijos que indiquen el tipo de dato real al que pertenecen, aunque nuevamente no los veremos hasta el *Tema 7: Variables y tipos de datos*

- **Literales lógicos: Los únicos literales lógicos válidos son true y false, que respectivamente representan los valores lógicos cierto y falso.**
- **Literales de carácter: Prácticamente cualquier carácter se puede representar encerrándolo entre comillas simples. Por ejemplo, 'a' (letra a), ' ' (carácter de espacio), '?' (símbolo de interrogación), etc. Las únicas excepciones a esto son los caracteres que se muestran en la Tabla 4.1, que han de representarse con secuencias de escape que indiquen su valor como código Unicode o mediante un formato especial tal y como se indica a continuación:**

Carácter	Código de escape Unicode	Código de escape especial
Comilla simple	\u0027	\'
Comilla doble	\u0022	\"
Carácter nulo	\u0000	\0
Alarma	\u0007	\a
Retroceso	\u0008	\b
Salto de página	\u000C	\f
Nueva línea	\u000A	\n
Retorno de carro	\u000D	\r

Tabulación horizontal	**\u0009**	**\t**
Tabulación vertical	**\u000B**	**\v**
Barra invertida	**\u005C**	****

Tabla 4.1: Códigos de escape especiales

En realidad, de la tabla anterior hay que matizar que el carácter de comilla doble también puede aparecer dentro de un literal de cadena directamente, sin necesidad de usar secuencias de escape. Por tanto, otros ejemplos de literales de carácter válidos serán '\?', '?', '\f', '\u0000', '\\', '\'', etc.

Aparte de para representar los caracteres de la tabla anterior, también es posible usar los códigos de escape Unicode para representar cualquier código Unicode, lo que suele usarse para representar literales de caracteres no incluidos en los teclados estándares.

Junto al formato de representación de códigos de escape Unicode ya visto, C# incluye un formato abreviado para representar estos códigos en los literales de carácter si necesidad de escribir siempre cuatro dígitos aún cuando el código a representar tenga muchos ceros en su parte izquierda. Este formato consiste en preceder el código de \x en vez de \u. De este modo, los literales de carácter '\U00000008', '\u0008', '\x0008', '\x008', '\x08' y '\x8' son todos equivalentes. Hay que tener en cuenta que este formato abreviado sólo es válido en los literales de carácter, y no a la hora de dar nombres a los identificadores.

- **Literales de cadena: Una cadena no es más que una secuencia de caracteres encerrados entre comillas dobles. Por ejemplo ?Hola, mundo?, ?camión?, etc. El texto contenido dentro estos literales puede estar formado por cualquier número de literales de carácter concatenados y sin las comillas simples, aunque si incluye comillas dobles éstas han de escribirse usando secuencias de escape porque si no el compilador las interpretaría como el final de la cadena.**

Aparte del formato de escritura de literales de cadenas antes comentado, que es el comúnmente utilizado en la mayoría de lenguajes de programación, C# también admite uno nuevo consistente en precederlos de un símbolo @, caso en que todo el contenido de la cadena sería interpretado tal cual, sin considerar la existencia de secuencias de escape. A este tipo de literales se les conoce como literales de cadena planos o literales verbatim y pueden incluso ocupar varias líneas. La siguiente tabla recoge algunos ejemplos de cómo se interpretan:

Literal de cadena	Interpretado como...
?Hola\tMundo?	**Hola Mundo**
@"Hola\tMundo?	**Hola\tMundo**
@?Hola Mundo?	**Hola Mundo**
@""""Hola Mundo""""	**"Hola Mundo"**

Tabla 4.2: Ejemplos de literales de cadena planos

El último ejemplo de la tabla se ha aprovechado para mostrar que si dentro de un literal de cadena plano se desea incluir caracteres de comilla doble sin que sean confundidos con el final de la cadena basta duplicarlos.

- **Literal nulo: El literal nulo es un valor especial que se representa en C# con la palabra reservada null y se usa como valor de las variables de objeto no inicializadas para así indicar que contienen referencias nulas.**

www.devjoker.com

Operadores

Un operador en C# es un símbolo formado por uno o más caracteres que permite realizar una determinada operación entre uno o más datos y produce un resultado.

A continuación se describen cuáles son los operadores incluidos en el lenguaje clasificados según el tipo de operaciones que permiten realizar, aunque hay que tener en cuenta que C# permite la redefinición del significado de la mayoría de los operadores según el tipo de dato sobre el que se apliquen, por lo que lo que aquí se cuenta se corresponde con los usos más comunes de los mismos:

- **Operaciones aritméticas**: Los operadores aritméticos incluidos en C# son los típicos de suma (+), resta (-), producto (*), división (/) y módulo (**%**) También se incluyen operadores de "menos unario" (–) y "más unario" (+)

  Relacionados con las operaciones aritméticas se encuentran un par de operadores llamados **checked** y **unchecked** que permiten controlar si se desea detectar los desbordamientos que puedan producirse si al realizar este tipo de operaciones el resultado es superior a la capacidad del tipo de datos de sus operandos. Estos operadores se usan así:

```
checked (<expresiónAritmética>)

unchecked(<expresiónAritmética>)
```

  Ambos operadores calculan el resultado de <expresiónAritmética> y lo devuelven si durante el cálculo no se produce ningún desbordamiento. Sin embargo, en caso de que haya desbordamiento cada uno actúa de una forma distinta: **checked** provoca un error de compilación si <expresiónAritmética> es una expresión constante y una excepción **System.OverflowException** si no lo es, mientras que **unchecked** devuelve el resultado de la expresión aritmética truncado para que quepa en el tamaño esperado.

  Por defecto, en ausencia de los operadores **checked** y **unchecked** lo que se hace es evaluar las operaciones aritméticas entre datos constantes como si se les aplicase **checked** y las operaciones entre datos no constantes como si se les hubiese aplicado **unchecked**.

- **Operaciones lógicas:** Se incluyen operadores que permiten realizar las operaciones lógicas típicas: "and" (**&&** y **&**), "or" (|| y |), "not" (**!**) y "xor" (^)

  Los operadores **&&** y || se diferencia de **&** y | en que los primeros realizan evaluación perezosa y los segundos no. La evaluación perezosa consiste en que si el resultado de evaluar el primer operando permite deducir el resultado de la operación, entonces no se evalúa el segundo y se devuelve dicho resultado directamente, mientras que la evaluación no perezosa consiste en evaluar siempre ambos operandos. Es decir, si el primer operando de una operación **&&** es falso se devuelve **false** directamente, sin evaluar el segundo; y si el primer operando de una || es cierto se devuelve **true** directamente, sin evaluar el otro.

- **Operaciones relacionales:** Se han incluido los tradicionales operadores de igualdad (==), desigualdad (**!=**), "mayor que" (>), "menor que" (<), "mayor o igual que" (>=) y "menor o igual que" (<=)

- **Operaciones de manipulación de bits**: Se han incluido operadores que permiten realizar a nivel de bits operaciones "and" (**&**), "or" (|), "not" (~), "xor" (^), desplazamiento a izquierda (<<) y desplazamiento a derecha (>>) El operador << desplaza a izquierda rellenando con ceros, mientras que el tipo de relleno realizado por >> depende del tipo de dato sobre el que se aplica: si es un dato con signo mantiene el signo, y en caso contrario rellena con ceros.

- **Operaciones de asignación:** Para realizar asignaciones se usa en C# el operador =, operador que además de realizar la asignación que se le solicita devuelve el valor asignado. Por ejemplo, la expresión a = b asigna a la variable a el valor de la variable b y devuelve dicho valor, mientras que la expresión c = a = b asigna a las variables c y a el valor de b (el operador = es asociativo por la derecha)

  También se han incluido operadores de asignación compuestos que permiten ahorrar tecleo a la hora de realizar asignaciones tan comunes como:

```
temperatura = temperatura + 15;   // Sin usar asignación compuesta
temperatura += 15;                // Usando asignación compuesta
```

  Las dos líneas anteriores son equivalentes, pues el operador compuesto += asigna a su primer operando el valor que tenía más el de su segundo operando (o sea, le suma el segundo operando) Como se ve, permite compactar bastante el código.

  Aparte del operador de asignación compuesto +=, también se ofrecen operadores de asignación compuestos para la mayoría de los operadores binarios ya vistos. Estos son: +=, -=, *=, /=, **%**=, **&**=, |=, ^=, <<= y >>=. Nótese que no hay versiones compuestas para los operadores binarios **&&** y ||.

  Otros dos operadores de asignación incluidos son los de incremento(++) y decremento (--) Estos operadores permiten, respectivamente, aumentar y disminuir en una unidad el valor de la variable sobre el que se aplican. Así, estas líneas de código son equivalentes:

```
temperatura = temperatura+1;//Sin usar asignación compuesta ni incremento
temperatura += 1;           //Usando asignación compuesta
temperatura++;              //Usando incremento
```

  Si el operador ++ se coloca tras el nombre de la variable (como en el ejemplo) devuelve el valor de la variable antes de incrementarla, mientras que si se coloca antes, devuelve el valor de ésta tras incrementarla; y lo mismo ocurre con el operador --. Por ejemplo:

```
c = b++; // Se asigna a c el valor de b y luego se incrementa b
c = ++b; // Se incrementa el valor de b y luego se asigna a c
```

  La ventaja de usar los operadores ++ y -- es que en muchas máquinas son más eficientes que el resto de formas de realizar sumas o restas de una unidad, pues el compilador puede traducirlos en una única instrucción en código máquina[5].

- **Operaciones con cadenas:** Para realizar operaciones de concatenación de cadenas se puede usar el mismo operador que para realizar sumas, ya que en

C# se ha redefinido su significado para que cuando se aplique entre operandos que sean cadenas o que sean una cadena y un carácter lo que haga sea concatenarlos. Por ejemplo, ?Hola?+? mundo? devuelve ?Hola mundo?, y ?Hola mund? + 'o' también.

- **Operaciones de acceso a tablas:** Una **tabla** es un conjunto de ordenado de objetos de tamaño fijo. Para acceder a cualquier elemento de este conjunto se aplica el operador postfijo **[]** sobre la tabla para indicar entre corchetes la posición que ocupa el objeto al que se desea acceder dentro del conjunto. Es decir, este operador se usa así:

  [<posiciónElemento>**]**

  Un ejemplo de su uso en el que se asigna al elemento que ocupa la posición 3 en una tabla de nombre tablaPrueba el valor del elemento que ocupa la posición 18 de dicha tabla es el siguiente:

```
tablaPrueba[3] = tablaPrueba[18];
```

  Las tablas se estudian detenidamente más adelante

- **Operador condicional:** Es el único operador incluido en C# que toma 3 operandos, y se usa así:

```
<condición> ? <expresión1> : <expresión2>
```

  El significado del operando es el siguiente: se evalúa <condición> Si es cierta se devuelve el resultado de evaluar <expresión1>, y si es falsa se devuelve el resultado de evaluar <condición2>. Un ejemplo de su uso es:

```
b = (a>0)? a : 0; // Suponemos a y b de tipos enteros
```

  En este ejemplo, si el valor de la variable a es superior a 0 se asignará a b el valor de a, mientras que en caso contrario el valor que se le asignará será 0.

  Hay que tener en cuenta que este operador es asociativo por la derecha, por lo que una expresión como a?b:c?d:e es equivalente a a?b:(c?d:e)

  No hay que confundir este operador con la instrucción condicional **if** que se tratará en el *Tema 8:Instrucciones*, pues aunque su utilidad es similar al de ésta, **?** devuelve un valor e **if** no.

- **Operaciones de delegados:** Un **delegado** es un objeto que puede almacenar en referencias a uno o más métodos y a través del cual es posible llamar a estos métodos. Para añadir objetos a un delegado se usan los operadores + y **+=**, mientras que para quitárselos se usan los operadores – y **-=**. Estos conceptos se estudiarán detalladamente en su correspondiente tema.

- **Operaciones de acceso a objetos:** Para acceder a los miembros de un objeto se usa el operador **.**, cuya sintaxis es:

```
<objeto>.<miembro>
```

  Si a es un objeto, ejemplos de cómo llamar a diferentes miembros suyos son:

```
a.b = 2;  // Asignamos a su propiedad a el valor 2
a.f();    // Llamamos a su método f()
a.g(2);   // Llamamos a su método g() pasándole como parámetro
          // el valor entero 2
```

  No se preocupe si no conoce los conceptos de métodos, propiedades, eventos y delegados en los que se basa este ejemplo, pues se explican detalladamente en temas posteriores.

- **Operaciones con punteros:** Un **puntero** es una variable que almacena una referencia a una dirección de memoria. Para obtener la dirección de memoria de un objeto se usa el operador **&**, para acceder al contenido de la dirección de memoria almacenada en un puntero se usa el operador *****, para acceder a un miembro de un objeto cuya dirección se almacena en un puntero se usa **->**, y para referenciar una dirección de memoria de forma relativa a un puntero se le aplica el operador **[]** de la forma puntero[desplazamiento]. Todos estos conceptos se explicarán más a fondo en el *Tema 18: Código inseguro*.

- **Operaciones de obtención de información sobre tipos:** De todos los operadores que nos permiten obtener información sobre tipos de datos el más importante es **typeof**, cuya forma de uso es:

```
typeof(<nombreTipo>)
```

  Este operador devuelve un objeto de tipo **System.Type** con información sobre el tipo de nombre <nombreTipo> que podremos consultar a través de los miembros ofrecidos por dicho objeto. Esta información incluye detalles tales como cuáles son sus miembros, cuál es su tipo padre o a qué espacio de nombres pertenece.

  Si lo que queremos es determinar si una determinada expresión es de un tipo u otro, entonces el operador a usar es **is**, cuya sintaxis es la siguiente:

```
<expresión> is <nombreTipo>
```

  El significado de este operador es el siguiente: se evalúa <expresión>. Si el resultado de ésta es del tipo cuyo nombre se indica en <nombreTipo> se devuelve **true**; y si no, se devuelve **false**. Como se verá en el *Tema 5: Clases*, este operador suele usarse en métodos polimórficos.

  Finalmente, C# incorpora un tercer operador que permite obtener información sobre un tipo de dato: **sizeof** Este operador permite obtener el número de

bytes que ocuparán en memoria los objetos de un tipo, y se usa así:

```
sizeof(<nombreTipo>)
```

sizeof sólo puede usarse dentro de código inseguro, que por ahora basta considerar que son zonas de código donde es posible usar punteros. No será hasta el *Tema 18: Código inseguro* cuando lo trataremos en profundidad.

Además, **sizeof** sólo se puede aplicar sobre nombres de tipos de datos cuyos objetos se puedan almacenar directamente en pila. Es decir, que sean estructuras (se verán en el *Tema 13*) o tipos enumerados (se verán en el *Tema 14*)

- **Operaciones de creación de objetos**: El operador más típicamente usado para crear objetos es **new**, que se usa así:

```
new <nombreTipo>(<parametros>)
```

Este operador crea un objeto de <nombreTipo> pasándole a su método constructor los parámetros indicados en <parámetros> y devuelve una referencia al mismo. En función del tipo y número de estos parámetros se llamará a uno u otro de los constructores del objeto. Así, suponiendo que a1 y a2 sean variables de tipo Avión, ejemplos de uso del operador **new** son:

```
Avión a1 = new Avión(); // Se llama al constructor sin parámetros
                        // de Avión
Avión a2 = new Avión("Caza");  // Se llama al constructor de Avión que
                               // toma como parámetro una cadena
```

En caso de que el tipo del que se haya solicitado la creación del objeto sea una clase, éste se creará en memoria dinámica, y lo que **new** devolverá será una referencia a la dirección de pila donde se almacena una referencia a la dirección del objeto en memoria dinámica. Sin embargo, si el objeto a crear pertenece a una estructura o a un tipo enumerado, entonces éste se creará directamente en la pila y la referencia devuelta por el **new** se referirá directamente al objeto creado. Por estas razones, a las clases se les conoce como **tipos referencia** ya que de sus objetos en pila sólo se almacena una referencia a la dirección de memoria dinámica donde verdaderamente se encuentran; mientras que a las estructuras y tipos enumerados se les conoce como **tipos valor** ya sus objetos se almacenan directamente en pila.

C# proporciona otro operador que también nos permite crear objetos. Éste es **stackalloc**, y se usa así:

```
stackalloc <nombreTipo>[<nElementos>]
```

Este operador lo que hace es crear en pila una tabla de tantos elementos de tipo <nombreTipo> como indique <nElementos> y devolver la dirección de memoria en que ésta ha sido creada. Por ejemplo:

```
int * p = stackalloc[100]; // p apunta a una tabla de 100 enteros.
```

stackalloc sólo puede usarse para inicializar punteros a objetos de tipos valor declarados como variables locales.

- **Operaciones de conversión:** Para convertir unos objetos en otros se utiliza el operador de conversión, que no consiste más que en preceder la expresión a convertir del nombre entre paréntesis del tipo al que se desea convertir el resultado de evaluarla. Por ejemplo, si l es una variable de tipo **long** y se desea almacenar su valor dentro de una variable de tipo **int** llamada i, habría que convertir previamente su valor a tipo **int** así:

```
i = (int) l;  // Asignamos a i el resultado de convertir el valor de l
              // a tipo int
```

Los tipos **int** y **long** están predefinidos en C# y permite almacenar valores enteros con signo. La capacidad de **int** es de 32 bits, mientras que la de **long** es de 64 bits. Por tanto, a no ser que hagamos uso del operador de conversión, el compilador no nos dejará hacer la asignación, ya que al ser mayor la capacidad de los **long**, no todo valor que se pueda almacenar en un **long** tiene porqué poderse almacenar en un **int**. Es decir, no es válido:

```
i = l; //ERROR: El valor de l no tiene porqué caber en i
```

Esta restricción en la asignación la impone el compilador debido a que sin ella podrían producirse errores muy difíciles de detectar ante truncamientos no esperados debido al que el valor de la variable fuente es superior a la capacidad de la variable destino.

Existe otro operador que permite realizar operaciones de conversión de forma muy similar al ya visto. Éste es el operador **as**, que se usa así:

```
<expresión> as <tipoDestino>
```

Lo que hace es devolver el resultado de convertir el resultado de evaluar <expresión> al tipo indicado en <tipoDestino> Por ejemplo, para almacenar en una variable p el resultado de convertir un objeto t a tipo tipo Persona se haría:

```
p = t as Persona;
```

Las únicas diferencias entre usar uno u otro operador de conversión son:

  - **as** sólo es aplicable a tipos referencia y sólo a aquellos casos en que existan conversiones predefinidas en el lenguaje. Como se verá más adelante, esto sólo incluye conversiones entre un tipo y tipos padres suyos y entre un tipo y tipos hijos suyos.

Una consecuencia de esto es que el programador puede definir cómo hacer conversiones de tipos por él definidos y otros mediante el operador **()**, pero no mediante **as**.Esto se debe a que **as** únicamente indica que se desea que una referencia a un objeto en memoria dinámica se trate como si el objeto fuese de otro tipo, pero no implica conversión ninguna. Sin embargo, **()** sí que implica conversión si el <tipoDestino> no es compatible con el tipo del objeto referenciado. Obviamente, el operador se aplicará mucho más rápido en los casos donde no sea necesario convertir.

- En caso de que se solicite hacer una conversión inválida **as** devuelve **null** mientras que **()** produce una excepción **System.InvalidCastException**.

www.devjoker.com

Instrucciones

Concepto de instrucción

Toda acción que se pueda realizar en el cuerpo de un método, como definir variables locales, llamar a métodos, asignaciones y muchas cosas más que veremos a lo largo de este tema, son **instrucciones**.

Las instrucciones se agrupan formando **bloques de instrucciones**, que son listas de instrucciones encerradas entre llaves que se ejecutan una tras otra. Es decir, la sintaxis que se sigue para definir un bloque de instrucciones es:

```
{
  <listaInstrucciones>
}
```

Toda variable que se defina dentro de un bloque de instrucciones sólo existirá dentro de dicho bloque. Tras él será inaccesible y podrá ser destruida por el recolector de basura. Por ejemplo, este código no es válido:

```
public void f();
{
    { int b; }
    b = 1;   // ERROR: b no existe fuera del bloque donde se declaró
}
```

Los bloques de instrucciones pueden anidarse, aunque si dentro de un bloque interno definimos una variable con el mismo nombre que otra definida en un bloque externo se considerará que se ha producido un error, ya que no se podrá determinar a cuál de las dos se estará haciendo referencia cada vez que se utilice su nombre en el bloque interno.

Instrucciones básicas

Definiciones de variables locales

En el *Tema 7: Variables y tipos de datos* se vio que las **variables locales** son variables que se definen en el cuerpo de los métodos y sólo son accesibles desde dichos cuerpos. Recuérdese que la sintaxis explicada para definirlas era la siguiente:

```
<modificadores> <tipoVariable> <nombreVariable> = <valor>;
```

También ya entonces se vio que podían definirse varias variables en una misma instrucción separando sus pares nombre-valor mediante comas, como en por ejemplo:

```
int a=5, b, c=-1;
```

Asignaciones

Una **asignación** es simplemente una instrucción mediante la que se indica un valor a almacenar en un dato. La sintaxis usada para ello es:

```
<destino> = <origen>;
```

En temas previos ya se han dado numerosos ejemplos de cómo hacer esto, por lo que no es necesario hacer ahora mayor hincapié en ello.

Llamadas a métodos

En el *Tema 8: Métodos* ya se explicó que una **llamada a un método** consiste en solicitar la ejecución de sus instrucciones asociadas dando a sus parámetros ciertos valores. Si el método a llamar es un método de objeto, la sintaxis usada para ello es:

```
<objeto>.<nombreMétodo>(<valoresParámetros>);
```

Y si el método a llamar es un método de tipo, entonces la llamada se realiza con:

```
<nombreTipo>.<nombreMétodo>(<valoresParámetros>);
```

Recuérdese que si la llamada al método de tipo se hace dentro de la misma definición de tipo donde el método fue definido, la sección <*nombreTipo*>**.** de la sintaxis es opcional.

Instrucción nula

La **instrucción nula** es una instrucción que no realiza nada en absoluto. Su sintaxis consiste en escribir un simple punto y coma para representarla. O sea, es:

```
;
```

Suele usarse cuando se desea indicar explícitamente que no se desea ejecutar nada, lo que es útil para facilitar la legibilidad del código o, como veremos más adelante en el tema, porque otras instrucciones la necesitan para indicar cuándo en algunos de sus bloques de instrucciones componentes no se ha de realizar ninguna acción.

www.devjoker.com

Estructuras de control

Las **instrucciones condicionales** son instrucciones que permiten ejecutar bloques de instrucciones sólo si se da una determinada condición. En los siguientes subapartados de este epígrafe se describen cuáles son las instrucciones condicionales disponibles en C#

Instrucción if

La **instrucción if** permite ejecutar ciertas instrucciones sólo si de da una determinada condición. Su sintaxis de uso es la sintaxis:

```
if (<condición>){
  <instruccionesIf>}
else{
   <instruccionesElse>}
```

El significado de esta instrucción es el siguiente: se evalúa la expresión <condición>, que ha de devolver un valor lógico. Si es cierta (devuelve **true**) se ejecutan las <instruccionesIf>, y si es falsa (**false**) se ejecutan las <instruccionesElse> La rama **else** es opcional, y si se omite y la condición es falsa se seguiría ejecutando a partir de la instrucción siguiente al **if**. En realidad, tanto <instruccionesIf> como <instruccionesElse> pueden ser una única instrucción o un bloque de instrucciones.

Un ejemplo de aplicación de esta instrucción es esta variante del HolaMundo:

```
using System;

class HolaMundoIf
{
    public static void Main(String[] args)
    {
         if (args.Length > 0){
           Console.WriteLine("Hola {0}!", args[0]);}
         else{
           Console.WriteLine("Hola mundo!");}
    }
}
```

Si ejecutamos este programa sin ningún argumento veremos que el mensaje que se muestra es ¡Hola Mundo!, mientras que si lo ejecutamos con algún argumento se mostrará un mensaje de bienvenida personalizado con el primer argumento indicado.

Instrucción switch

La **instrucción switch** permite ejecutar unos u otros bloques de instrucciones según el valor de una cierta expresión. Su estructura es:

```
switch (<expresión>)
{
  case <valor1>:  <bloque1>
   <siguienteAcción>
  case <valor2>:  <bloque2>
   <siguienteAcción>
  ...
  default:  <bloqueDefault>
   <siguienteAcción>
}
```

El significado de esta instrucción es el siguiente: se evalúa . Si su valor es se ejecuta el , si es se ejecuta , y así para el resto de valores especificados. Si no es igual a ninguno de esos valores y se incluye la rama **default**, se ejecuta el ; pero si no se incluye se pasa directamente a ejecutar la instrucción siguiente al **switch**.

Los valores indicados en cada rama del **switch** han de ser expresiones constantes que produzcan valores de algún tipo básico entero, de una enumeración, de tipo **char** o de tipo **string**. Además, no puede haber más de una rama con el mismo valor.

En realidad, aunque todas las ramas de un **switch** son opcionales siempre se ha de incluir al menos una. Además, la rama **default** no tiene porqué aparecer la última si se usa, aunque es recomendable que lo haga para facilitar la legibilidad del código.

El elemento marcado como <siguienteAcción> colocado tras cada bloque de instrucciones indica qué es lo que ha de hacerse tras ejecutar las instrucciones del bloque que lo preceden. Puede ser uno de estos tres tipos de instrucciones:

```
goto case <valori>;

goto default;

break;
```

Si es un **goto case** indica que se ha de seguir ejecutando el bloque de instrucciones asociado en el **switch** a la rama del <valori> indicado, si es un **goto default** indica que se ha de seguir ejecutando el bloque de instrucciones de la rama **default**, y si es un **break** indica que se ha de seguir ejecutando la instrucción siguiente al switch.

El siguiente ejemplo muestra cómo se utiliza **switch**:

```
using System;
class HolaMundoSwitch
{
   public static void Main(String[] args)
  {
      if (args.Length > 0)
        switch(args[0])
        {
         case "José":
              Console.WriteLine("Hola José. Buenos días");
              break;
         case "Paco":
              Console.WriteLine("Hola Paco. Me alegro de verte");
              break;
         default:
              Console.WriteLine("Hola {0}", args[0]);
              break;
        }
     else
        Console.WriteLine("Hola Mundo");
  }
}
```

Este programa reconoce ciertos nombres de personas que se le pueden pasar como argumentos al lanzarlo y les saluda de forma especial. La rama **default** se incluye para dar un saludo por defecto a las personas no reconocidas.

Para los programadores habituados a lenguajes como C++ es importante resaltarles el hecho de que, a diferencia de dichos lenguajes, C# obliga a incluir una sentencia **break** o una sentencia **goto case** al final de cada rama del **switch** para evitar errores comunes y difíciles de detectar causados por olvidar incluir **break;** al final de alguno de estos bloques y ello provocar que tras ejecutarse ese bloque se ejecute también el siguiente.

Instrucciones iterativas

Las **instrucciones iterativas** son instrucciones que permiten ejecutar repetidas veces una instrucción o un bloque de instrucciones mientras se cumpla una condición. Es decir, permiten definir bucles donde ciertas instrucciones se ejecuten varias veces. A continuación se describen cuáles son las instrucciones de este tipo incluidas en C#.

Instrucción while

La **instrucción while** permite ejecutar un bloque de instrucciones mientras se de una cierta instrucción. Su sintaxis de uso es:

```
while (<condición>)
{
  <instrucciones>
}
```

Su significado es el siguiente: Se evalúa la <condición> indicada, que ha de producir un valor lógico. Si es cierta (valor lógico **true**) se ejecutan las <instrucciones> y se repite el proceso de evaluación de <condición> y ejecución de <instrucciones> hasta que deje de serlo. Cuando sea falsa (**false**) se pasará a ejecutar la instrucción siguiente al **while**. En realidad <instrucciones> puede ser una única instrucción o un bloque de instrucciones.

Un ejemplo cómo utilizar esta instrucción es el siguiente:

```
using System;
class HolaMundoWhile
{
    public static void Main(String[] args)
    {
       int actual = 0;
```

```
        if (args.Length > 0)
            while (actual < args.Length)
            {
             Console.WriteLine("¡Hola {0}!", args[actual]);
             actual  = actual + 1;
            }
        else
            Console.WriteLine("¡Hola mundo!");
    }
}
```

En este caso, si se indica más de un argumento al llamar al programa se mostrará por pantalla un mensaje de saludo para cada uno de ellos. Para ello se usa una variable actual que almacena cuál es el número de argumento a mostrar en cada ejecución del **while**. Para mantenerla siempre actualizada lo que se hace es aumentar en una unidad su valor tras cada ejecución de las <instrucciones> del bucle.

Por otro lado, dentro de las <instrucciones> de un **while** pueden utilizarse las siguientes dos instrucciones especiales:

- **break;**: Indica que se ha de abortar la ejecución del bucle y continuarse ejecutando por la instrucción siguiente al **while**.
- **continue;**: Indica que se ha de abortar la ejecución de las <instrucciones> y reevaluarse la <condición> del bucle, volviéndose a ejecutar las <instrucciones> si es cierta o pasándose a ejecutar la instrucción siguiente al **while** si es falsa.

Instrucción do...while

La instrucción **do...while** es una variante del **while** que se usa así:

```
do {
    <instrucciones>
} while(<condición>);
```

La única diferencia del significado de **do...while** respecto al de **while** es que en vez de evaluar primero la condición y ejecutar <instrucciones> sólo si es cierta, **do...while** primero ejecuta las <instrucciones> y luego mira la <condición> para ver si se ha de repetir la ejecución de las mismas. Por lo demás ambas instrucciones son iguales, e incluso también puede incluirse **break;** y **continue;** entre las <instrucciones> del **do...while**.

do ... while está especialmente destinado para los casos en los que haya que ejecutar las <instrucciones> al menos una vez aún cuando la condición sea falsa desde el principio, como ocurre en el siguiente ejemplo:

```
using System;
class HolaMundoDoWhile
{
  public static void Main()
  {
      String leído;
      do
      {
         Console.WriteLine("Clave: ");
         leído = Console.ReadLine();
      }
      while (leído != "José");
         Console.WriteLine("Hola José");
  }
}
```

Este programa pregunta al usuario una clave y mientras no introduzca la correcta (José) no continuará ejecutándose. Una vez que introducida correctamente dará un mensaje de bienvenida al usuario.

Instrucción for

La **instrucción for** es una variante de **while** que permite reducir el código necesario para escribir los tipos de bucles más comúnmente usados en programación. Su sintaxis es:

```
for (<inicialización>; <condición>; <modificación>){
 <instrucciones>
 }
```

El significado de esta instrucción es el siguiente: se ejecutan las instrucciones de <inicialización>, que suelen usarse para definir e inicializar variables que luego se usarán en <instrucciones>. Luego se evalúa <condición>, y si es falsa se continúa ejecutando por la instrucción siguiente al **for**; mientras que si es cierta se ejecutan las <instrucciones> indicadas, luego se ejecutan las instrucciones de <modificación> -que como su nombre indica suelen usarse para

modificar los valores de variables que se usen en <instrucciones>- y luego se reevalúa <condición> repitiéndose el proceso hasta que ésta última deje de ser cierta.

En <inicialización> puede en realidad incluirse cualquier número de instrucciones que no tienen porqué ser relativas a inicializar variables o modificarlas, aunque lo anterior sea su uso más habitual. En caso de ser varias se han de separar mediante comas (**,**), ya que el carácter de punto y coma (**;**) habitualmente usado para estos menesteres se usa en el **for** para separar los bloques de <inicialización>, <condición> y <modificación> Además, la instrucción nula no se puede usar en este caso y tampoco pueden combinarse definiciones de variables con instrucciones de otros tipos.

Con <modificación> pasa algo similar, ya que puede incluirse código que nada tenga que ver con modificaciones pero en este caso no se pueden incluir definiciones de variables.

Como en el resto de instrucciones hasta ahora vistas, en <instrucciones> puede ser tanto una única instrucción como un bloque de instrucciones. Además, las variables que se definan en <inicialización> serán visibles sólo dentro de esas <instrucciones>.

La siguiente clase es equivalente a la clase HolaMundoWhile ya vista solo que hace uso del **for** para compactar más su código:

```
using System;
class HolaMundoFor
{
    public static void Main(String[] args)
    {
       if (args.Length > 0)
         for (int actual = 0; actual < args.Length; actual++) {
             Console.WriteLine("¡Hola {0}!", args[actual]);
         }
       else
         Console.WriteLine("¡Hola mundo!");
    }
}
```

Al igual que con **while**, dentro de las <instrucciones> del **for** también pueden incluirse instrucciones **continue;** y **break;** que puedan alterar el funcionamiento normal del bucle.

Instrucción foreach

La **instrucción foreach** es una variante del **for** pensada especialmente para compactar la escritura de códigos donde se realice algún tratamiento a todos los elementos de una colección, que suele un uso muy habitual de **for** en los lenguajes de programación que lo incluyen. La sintaxis que se sigue a la hora de escribir esta instrucción **foreach** es:

```
foreach (<tipoElemento> <elemento> in <colección>) {
   <instrucciones>
}
```

El significado de esta instrucción es muy sencillo: se ejecutan <instrucciones> para cada uno de los elementos de la <colección> indicada. <elemento> es una variable de sólo lectura de tipo <tipoElemento> que almacenará en cada momento el elemento de la colección que se esté procesando y que podrá ser accedida desde <instrucciones>.

Es importante señalar que <colección> no puede valer **null** porque entonces saltaría una excepción de tipo **System.NullReferenceException**, y que <tipoElemento> ha de ser un tipo cuyos objetos puedan almacenar los valores de los elementos de <colección>

En tanto que una tabla se considera que es una colección, el siguiente código muestra cómo usar **for** para compactar aún más el código de la clase HolaMundoFor anterior:

```
using System;
class HolaMundoForeach
{
    public static void Main(String[] args)
    {
       if (args.Length > 0)
          foreach(String arg in args) {
           Console.WriteLine("¡Hola {0}!", arg);
          }
       else
          Console.WriteLine("¡Hola mundo!");
    }
}
```

Las tablas multidimensionales también pueden recorrerse mediante el **foreach**, el cual pasará por sus elementos en orden tal y como muestra el siguiente fragmento de código:

```
int[,] tabla = { {1,2}, {3,4} };
foreach (int elemento in tabla)
    Console.WriteLine(elemento);
```

Cuya salida por pantalla es:

```
1
2
3
4
```

En general, se considera que una colección es todo aquel objeto que implemente las interfaces **IEnumerable** o **IEnumerator** del espacio de nombres **System.Collections** de la BCL, que están definidas como sigue:

```
interface IEnumerable
{
  [C#]
  IEnumerator GetEnumerator();
}

interface IEnumerator
{
  [C#]
  object Current {get;}
  [C#]
  bool MoveNext();
  [C#]
  void Reset();
}
```

El método **Reset()** ha de implementarse de modo que devuelva el enumerador reiniciado a un estado inicial donde aún no referencie ni siquiera al primer elemento de la colección sino que sea necesario llamar a **MoveNext()** para que lo haga.

El método **MoveNext()** se ha de implementar de modo que haga que el enumerador pase a apuntar al siguiente elemento de la colección y devuelva un booleano que indique si tras avanzar se ha alcanzado el final de la colección.

La propiedad **Current** se ha de implementar de modo que devuelva siempre el elemento de la colección al que el enumerador esté referenciando. Si se intenta leer **Current** habiéndose ya recorrido toda la colección o habiéndose reiniciado la colección y no habiéndose colocado en su primer elemento con **MoveNext()**, se ha de producir una excepción de tipo **System.Exception.SystemException.InvalidOperationException**

Otra forma de conseguir que **foreach** considere que un objeto es una colección válida consiste en hacer que dicho objeto siga el **patrón de colección**. Este patrón consiste en definir el tipo del objeto de modo que sus objetos cuenten con un método público **GetEnumerator()** que devuelva un objeto no nulo que cuente con una propiedad pública llamada **Current** que permita leer el elemento actual y con un método público **bool MoveNext()** que permita cambiar el elemento actual por el siguiente y devuelva **false** sólo cuando se haya llegado al final de la colección.

El siguiente ejemplo muestra ambos tipos de implementaciones:

```
using System;
using System.Collections;

class Patron
{
 private int actual = -1;
 public Patron GetEnumerator()
 {
  return this;
 }

 public int Current
 {
  get {return actual;}
 }

 public bool MoveNext()
 {
  bool resultado = true;
  actual++;
  if (actual==10)
      resultado = false;
  return resultado;
 }
```

```
}

class Interfaz:IEnumerable,IEnumerator
{
 private int actual = -1;
 public object Current
 {
  get {return actual;}
 }

 public bool MoveNext()
 {
   bool resultado = true;
   actual++;
   if (actual==10)
     resultado = false;
   return resultado;
 }

 public IEnumerator GetEnumerator()
 {
   return this;
 }

 public void Reset()
 {
   actual = -1;
 }
}

class Principal
{
 public static void Main()
 {
    Patron obj = new Patron();
    Interfaz obj2 = new Interfaz();
    foreach (int elem in obj)
       Console.WriteLine(elem);
    foreach (int elem in obj2)
       Console.WriteLine(elem);
 }
}
```

Nótese que en realidad en este ejemplo no haría falta implementar **IEnumerable**, puesto que la clase Interfaz ya implementa **IEnumerator** y ello es suficiente para que pueda ser recorrida mediante **foreach**.

La utilidad de implementar el patrón colección en lugar de la interfaz **IEnumerable** es que así no es necesario que **Current** devuelva siempre un **object**, sino que puede devolver objetos de tipos más concretos y gracias a ello puede detectarse al compilar si el <tipoElemento> indicado puede o no almacenar los objetos de la colección.

Por ejemplo, si en el ejemplo anterior sustituimos en el último **foreach** el <tipoElemento> indicado por Patrón, el código seguirá compilando pero al ejecutarlo saltará una excepción **System.InvalidCastException**. Sin embargo, si la sustitución se hubiese hecho en el penúltimo **foreach**, entonces el código directamente no compilaría y se nos informaría de un error debido a que los objetos **int** no son convertibles en objetos Patrón.

También hay que tener en cuenta que la comprobación de tipos que se realiza en tiempo de ejecución si el objeto sólo implementó la interfaz **IEnumerable** es muy estricta, en el sentido de que si en el ejemplo anterior sustituimos el <tipoElemento> del último **foreach** por **byte** también se lanzará la excepción al no ser los objetos de tipo **int** implícitamente convertibles en **byte**s sino sólo a través del operador **()** Sin embargo, cuando se sigue el patrón de colección las comprobaciones de tipo no son tan estrictas y entonces sí que sería válido sustituir **int** por **byte** en <tipoElemento>.

El problema de sólo implementar el patrón colección es que este es una característica propia de C# y con las instrucciones **foreach** (o equivalentes) de lenguajes que no lo soporten no se podría recorrer colecciones que sólo siguiesen este patrón. Una solución en estos casos puede ser hacer que el tipo del objeto colección implemente tanto la interfaz **IEnumerable** como el patrón colección. Obviamente esta interfaz debería implementarse explícitamente para evitarse conflictos derivados de que sus miembros tengan signaturas coincidentes con las de los miembros propios del patrón colección.

Si un objeto de un tipo colección implementa tanto la interfaz **IEnumerable** como el patrón de colección, entonces en C# **foreach** usará el patrón colección para recorrerlo.