Introducción al lenguaje y características derivadas de su entorno de ejecución.

A lo largo de los últimos meses, hemos estado hablando de algunas características del nuevo entorno .NET Framework que Microsoft está desarrollando y a medida que pasaba el tiempo, aquí en **Grupo EIDOS**, nos hemos ido dividiendo los roles de trabajo, para poder abarcar todo este nuevo paradigma de la programación con la profundidad que se merece. Así pues, tras unos primeros devaneos con Visual Studio, el nuevo Visual Basic .NET y su interfaz de usuario (ver artículo del mes de Abril), cedo los trastos de Visual Basic a mi compañero **Luis Miguel Blanco**, de sobra conocido por los lectores de Algoritmo y de La Librería Digital, para dedicarme por entero al nuevo lenguaje de programación: **C#** (C-Sharp).

Pretendemos, a lo largo de los próximos meses, ir publicando un curso completo del lenguaje que sirva de introducción a nivel básico-intermedio para los lectores, de forma que, cuando hacia el mes de Noviembre aparezca el producto, ya dispongan de una buena base para poder trabajar y continuar la producción con este lenguaje (como quizá sepa ya el lector, el pasado día 21 de Junio, coincidiendo con el primer evento Tech-Ed del año, celebrado en Atlanta (Georgia), se hizo, por fin pública la beta 2 de Microsoft .NET Framework, así como -para usuarios del MSDN y otros- la beta 2 de Visual Studio .NET. A título personal, tengo fundadas razones para dudar de la demora hasta Febrero/2002, como se ha comentado en algunos foros, y por si esto fuera poco, el **Dr.GUI**, mentor ficticio del MSDN de Microsoft afirmaba en su 6ª entrega de sus cursos sobre .NET que "a partir del próximo número, todo el código será compatible con la Beta 2 de .NET Framework, puesto que **Microsoft no planea ya ningún cambio significativo al API entre la Beta 2 y el producto final, solamente ajustes, optimización y algún cambio de mínima importancia**", lo que da idea de lo avanzado del proyecto).

Introducción a C#

El lenguaje C# se presenta como el último invento en materia de lenguajes de programación, y constituye también la más reciente y ambiciosa apuesta en este sentido por parte de *Microsoft*. Quizás, lo primero que habría que aclarar, es que, de todo el **.NET Framework**, es ***la única parte que puede considerarse terminada***, hasta el punto de que el propio Visual Studio .NET ha sido construido al 90% en **C#** y el 10% restante en **C++**. Por otro lado, el lenguaje merece el calificativo de estándar, en el sentido de que –al igual que algunos otros aspectos del entorno- está siendo sometido a estandarización por parte de **ECMA**, la misma entidad de normalización que llevó a cabo la estandarización de **Javascript.**

Nota: ***En una reciente visita Madrid por parte de dos de los promotores y desarrolladores de .NET (Ari Bixhorn y Drew Fletcher), a la que tuvimos ocasión de asistir, se respondía a una pregunta del amigo Francisco Charte en ese sentido, afirmando que ellos eran conscientes de que ECMA estaba manteniendo reuniones mensuales con los equipos de estudio de la normalización (los Working Teams), y que esperaban que dicha labor estuviera completada, sino antes, sí para el momento de la aparición del producto.***

Rumores sobre el impacto de C# en el mundo de la programación

Es natural que, ante la creciente comunidad mundial de profesionales de las TI, la aparición de un nuevo lenguaje tenga más impacto hoy que en etapas anteriores, más arcaicas. Los medios lo favorecen, y la difusión de Internet, también. Por eso, nos hemos paseado por algunas revistas en la Web donde los gurús mas conocidos dan sus opiniones.

Anders Hejlsberg, arquitecto principal del lenguaje C#

La impresión de estos autores podríamos resumirla así: C# supone una mejora respecto a otros lenguajes existentes por dos razones básicas: primero, por que es el último, y por lo tanto, el más adaptado a las necesidades actuales del programador y el que más ha *aprendido* de los demás, heredando lo mejor de cada entorno, y añadiendo las cosas que los programadores solicitaban. En segundo término, por que su creador principal (el jefe del equipo de desarrollo, el danés **Anders Hejlsberg**, de quien Bixhorn y Fletcher hablaban maravillas), ha podido hacer un excelente trabajo basado su experiencia personal (es el diseñador de **Turbo Pascal 5.5** y **Delphi**), con tiempo suficiente, y un pequeño pero impresionante equipo de colaboradores entre los que figuran Peter Kukol, Jeffrey Richter, etc.

De hecho, el propio nombre del lenguaje (se pronuncia **CSharp**) fue una decisión posterior, como se ha sabido, en el sentido de que era una extensión de C++: C++++ (con 4 +), para indicar su origen principal. Si bien esto es cierto, no lo es menos que en un famoso e-mail que se hizo público a raíz del contencioso Sun-Microsoft, dirigido por Hejlsberg a Bill Gates, se habla del proyecto de desarrollo del nuevo lenguaje con frases esclarecedoras sobre la intención con que se construía...

Así, podemos leer (la traducción es mía) "**Peter Kukol y yo hemos estado trabajando en un conjunto de extensiones de Java que yo definiría como "un cuidadoso maridaje entre Java y C**" (Original: "***Peter Kukol and I have been working on a set of Java language extensions that I would characterize as "A careful marriage of Java and C.***"). Y más adelante, en una significativa referencia al nombre del nuevo engendro, afirma "**A falta de un nombre mejor, estamos llamando al lenguaje J++**" (Original: **For lack of a better name we're calling the language J++**), en una clara alusión a sus fundamentos basados también en el lenguaje Java. Finalmente, otra frase entresacada de este e-mail, explica claramente el propósito del diseño al afirmar que "**lo que muchos programadores quieren es una versión "limpia" de C++ que aporte los beneficios de productividad de Java combinados con las características de bajo nivel de C"** (Original: ***what a lot of programmers want is a "cleaned up: version of C++ which gives them the productivity benefits of Java coupled with the lower level features of C***").

Naturalmente, las reacciones han sido muy variadas: desde aquellos que –por el mero hecho de que el producto sea de **Microsoft**- lo desdeñan, hasta (una mayoría), que aprecian sus nuevas características en lo que valen y promueven su utilización. Quizá por eso, sean de especial valor las opiniones de algunos *gurús (Ben Albahari de Genamics, Christopher P. Calisi de SD Times,* e incluso *Jim Farley* de O'Really), que estando dedicados a la programación en Java no les importa reconocer en sus comparativas (el lector tiene las referencias bibliográficas al final de este articulo), que "**si eres un seguidor acérrimo de Java...** (el utiliza la palabra *evangelist*, que tiene otras connotaciones en inglés) **no te fíes de que C# sea simplemente otro marketing de Microsoft. Mejor, prepárate**."

Bien, pues vamos a prepararnos. Sobre todo teniendo en cuenta que, como se repite con más frecuencia cada vez, éste cambio a .NET va a afectar a la forma de programar al menos por los próximos 10 años...

El lenguaje C# y el Entorno Común de Ejecución (CLR)

Una de las características principales de C# es que se trata de un lenguaje que compila (por defecto) a un formato intermedio, al estilo de Java, denominado **Intermediate Language (IL)**, que posteriormente, debe de ser interpretado por un entorno de ejecución, una máquina JIT (*just-in-time*), también al estilo de Java. La gran diferencia respecto a Java es que, ése intérprete será común a todos los lenguaje soportados por el entorno de ejecución (veintitantos ya...) y mediante este mecanismo permitirá que los componentes realizados en cualquier lenguaje puedan comunicarse entre sí.

Se trata pues, de **una extensión del concepto inicial que dio origen a Java: en lugar de un único lenguaje para muchas plataformas, se pretende un entorno común multiplataforma, que soporte muchos lenguajes, basándose en que todos ellos compilen a un mismo código intermedio.** Para hacer viable esta idea, se ha optimizado considerablemente la velocidad, respecto a Java y ya se están anunciando los primeros .NET Framework para otras plataformas: El pasado mes de Marzo, Steve Ballmer anunciaba la disponibilidad para **Linux**, y están en marcha las implementaciones para **Unix**, **McIntosh System-8** y **BEos**.

Este lenguaje intermedio, es gestionado por un mecanismo llamado Entorno Común de Ejecución (**Common Language Runtime**), encargado, además, de la gestión de memoria, y en general, de las tareas más importantes, relacionadas con la ejecución de programas.

Programación en C#

Para una primera toma de contacto con el lenguaje, nada mejor que ponernos manos a la obra e ir analizando algunos casos prácticos, empezando por el ya clásico "Hola Mundo".

Principios básicos de la construcción de programas en C#

Pero antes, recordemos las bases sobre las que se asienta la construcción de un programa escrito en C#:

1) Todo programa compilado para el entorno común de ejecución (CLR) comparte un mismo código intermedio (*Intermediate Language* ó IL) que debe ser ejecutado por un intérprete JIT que resida en la plataforma de ejecución. Aunque pueden distinguirse entre componentes y ejecutables, en éste último caso, el formato resultante es independiente del lenguaje que se utilizó en su escritura. El formato de los ejecutables independientes se denomina PE (*Portable Executable*).

2) El entorno común de ejecución (CLR) es invocado por cualquier programa escrito en el formato PE, y es quien se hace cargo del mantener la zona de memoria en la que se ejecuta el programa y suministra un *sistema de tipos de datos común* (*Common Type System)*, que describe los tipos soportados por el intérprete y cómo estos tipos pueden interactuar unos con otros y cómo puede ser almacenados en metadatos.

3) Los servicios del sistema operativo se suministran mediante librerías (DLL) dispuestas por el CLR cuando se instala **.NET Framework** en una plataforma (*Windows*, *McIntosh*, *Linux*, etc.). Dichas librerías son referenciadas en el código de C# mediante la sentencia **using** seguida del nombre de la librería deseada. Tales declaraciones reciben el nombre de Espacios Calificados o Espacios con Nombre (***Namespaces***).

4) En C#, todo puede ser tratado como un objeto, aunque no obligatoriamente, y sin la penalización que supone esta característica en lenguajes orientados a objetos "puros", como SmallTalk. Por tanto cualquier código en C# estará incluido en una clase, ya sea esta instanciada o no. Además, se protege la inversión realizada previamente en desarrollo, ya que permite la llamada a componentes COM, a librerías nativas del API de Windows (DLL's), permite el acceso de bajo nivel cuando sea apropiado y los ejecutables y componentes producidos con él se integran perfectamente con otros ejecutables o componentes realizados en otros lenguajes del entorno.

5) La sintaxis de C# recuerda bastante la de Java, si bien los conceptos implicados en la construcción de objetos, tipos, propiedades, métodos, y eventos no suelen ser idénticos teniendo -en ocasiones- notables diferencias conceptuales. La mayor

similitud se encuentra, como cabía esperar, en el tratamiento de las estructuras de control del programa (prácticamente idénticas a C++ y Java).

Primer ejemplo: Hola mundo desde C#

Siguiendo la tradición, comentaremos para empezar el código de la implementación de una salida por la consola con el texto "Hola Mundo". El código es el siguiente:

Código fuente del "Hola Mundo"

```
class HolaMundoCS

{

    public static void Main()

    {

         Console.WriteLine("Hola desde C#");

    }
```

Y, como es de esperar, produce una salida en una ventana del DOS (Consola), con el texto indicado entre paréntesis por el método WriteLine ("Hola desde C#").

Análisis del código

Vamos a analizar el código fuente para irnos familiarizando con los mecanismos típicos de un programa en C#: La primera declaración indica que vamos a utilizar el espacio calificado (o **NameSpace**) llamado **System.** *Un espacio calificado o nominado es una especie de firma digital con la que podemos hacer referencia a una librería o bien -mediante la declaración* ***NameSpace****- hacer que sirva de identificador de todos los módulos que pertenezcan a una aplicación, o un fragmento de aplicación.* **System** es el NameSpace básico para todas las aplicaciones que requieran algún servicio del sistema (que son prácticamente todas). Note el lector que la declaración del espacio calificado podría haberse eliminado si en el código hubiéramos hecho referencia a la jerarquía completa, esto es, si hubiéramos escrito:

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

Más sobre el concepto de Namespace

Esto es así, porque el CLR pone a disposición de cualquier aplicación todos los servicios de sistema operativo a través de un conjunto jerárquico de clases con sólo hacer referencia a ellas. No obstante, y para facilitar la escritura del código utilizamos el concepto de **Namespace** (*espacio calificado ó espacio con nombre*). Un **namespace** es una forma adecuada de organizar clases y otros tipos de datos en una jerarquía. Podríamos decir que sirven al mismo tiempo para 3 propósitos: organizar (dentro de la jerarquía), para firmar (o marcar de forma única) fragmentos de código como pertenecientes a ese **namespace**, evitando colisiones con otros fragmentos que puedan contener definiciones iguales, y como método de abreviación, ya que una vez declarado el **namespace** podemos referirnos a sus contenidos en el código sin necesidad de repetir toda la secuencia jerárquica, tal y como aparece en la línea de código anterior.

Los espacios calificados pueden estar predefinidos, como en este caso, o ser declarados en el código por el propio usuario. Como nuestra aplicación utiliza una única salida de texto por el dispositivo por defecto (Consola), no es preciso hacer ninguna referencia a Windows ni a sus servicios. La consola está representada en dicho espacio por el objeto **Console** que dispone de métodos para escribir texto en la consola por defecto **(Write()** y **Writeline())** y también para recoger entradas de datos del mismo dispositivo (**Read()** y **Readline()**) (Piense el lector, que puede haber casos en los que la labor de un programa es totalmente desatendida (esto es, no dispone de ninguna interfaz de usuario), ya que sus salidas no van dirigidas a un usuario en concreto, sino a un fichero, tal y como ocurre muchas veces con los componentes).

Main(): la función principal

La segunda línea contiene otra declaración importante. *Como se ha indicado, todo el código de un programa en C# debe estar incluido en una clase*. Además, en C# no existen métodos ni variables globales, por tanto, mediante la palabra reservada **class** declaramos la clase *HolaMundoCS*, que contiene todo el código ejecutable.

Esta clase está compuesta por una única función, que, en éste caso, hace las veces de método constructor y función principal (***Main***). Dicha función esta precedida de 3 calificadores: de ámbito (**public**), de acceso (**static**) y de valor de retorno (**void**) y es la función principal de todo programa ejecutable en C#. En ella es donde comienza y termina el control del programa y es donde se crean objetos y se ejecutan otros métodos.

También debemos tener en cuenta que **Main**() admite varios modos de implementación similares según devuelva o no valores (en nuestro ejemplo anterior, el calificador void indica que no devuelve ninguno), y también es posible eliminar la declaración public, haciendo que no se pueda acceder a el desde otras clases. Finalmente, se puede declarar un ***array*** de argumentos, de manera que recoja los argumentos que se le pasen al programa en la línea de comandos.

Versión del "Hola Mundo" que devuelve un "int"

```
class HolaMundoCS

{

    public static int Main()

    {

          Console.WriteLine("Hola desde C#");

          return 0;

    }
```

Versión del "Hola Mundo" que puede recibir argumentos en la línea de comandos

```
class HolaMundoCS

{

    public static void Main ( string[] args )

    {

          Console.WriteLine("Hola desde C#");

    }
```

Es especialmente importante distinguir desde el principio estos calificadores (sobre todo, los dos primeros) por que establecen la manera en que el código que escribimos puede ser utilizado. Aunque, a continuación incluimos unas tablas resumen del modo de funcionamiento de todos los calificadores importantes del lenguaje, aclaremos ante todo (sobre todo para los poco versados en éste tipo de lenguajes) que ***el código incluido en una clase puede ser accedido básicamente de dos formas: mediante un objeto que haga referencia (instancie) dicha clase, o de forma directa, al estilo de como lo hacemos con las funciones de librería en otros lenguajes, sin utilizar ningún objeto explícito***.

Clases, métodos y propiedades de tipo static

En éste segundo caso, la clase o el método al que queramos referirnos en ésta forma será declarado como **static**. A una clase definida de esta forma, se la denomina clase global. La jerarquía de objetos de .NET contiene muchas clases de este tipo, no pudiendo heredarse de ellas, y estando pensadas para ser usadas directamente. El método inicial de toda aplicación se declara como **static** precisamente para que el **CLR** pueda acceder a él sin necesidad de haberse instanciado ningún objeto. Cuando un método se declara como **static**, no es posible añadir los calificadores que permiten heredar de el (**virtual**, **abstract**) o sobrescribir su modo de implementación (**override**). Lo mismo es aplicable para las propiedades declaradas como **static**. Tampoco se permite la utilización dentro de ese código de la palabra reservada que se utiliza para referirse a la propia clase (**this**).

Entradas y salidas de datos

En principio, el lenguaje C# no dispone de forma nativa de ningún mecanismo de acceso directo a los dispositivos periféricos (consola, teclado, etc). Dichos accesos vienen suministrados mediante librerías dentro del Microsoft .NET Framework. En el caso de nuestro programa inicial, dicho acceso se obtiene mediante System.Console como hemos comentado, (que tambien da acceso al dispositivo de errores estándar (standard error). Si la aplicación requiere ventanas de Windows (lo más normal), sería preciso hacer referencia a la librería System.WinForms, y así sucesivamente.

Los procesos de Edición, Compilación y/o llamada al ejecutable.

Por otro lado, y debido a su naturaleza, el lenguaje C# no requiere de ningún entorno de desarrollo, estrictamente hablando. Basta con un editor de texto y una llamada al compilador pasándole los parámetros de compilación adecuados. Podemos utilizar el Bloc de Notas, o cualquiera de los editores Shareware disponibles en la red, para este propósito. El editor Antechinus es un buen ejemplo de ello. Para compilar un programa escrito en C#, podemos utilizar dos mecanismos, dependiendo del editor que estemos utilizando. Si el editor es el propio de Visual Studio .NET, basta con seleccionar la opción de construir la aplicacion (Build). Pero no olvidemos que una llamada directa al compilador puede hacer la misma labor.

Para llamar al compilador de C# desde la línea de comandos, teclearíamos:

```
csc Hola.cs /out:Hola.exe
```

donde csc es el nombre del compilador, "Hola.cs" nuestro fichero de código fuente en C#, y "Hola.exe" el nombre del fichero de salida. En caso de que se requiera la presencia de librerías adicionales (por defecto se incluye mscorelib.dll, que es la que contiene a System), haríamos las referencias añadiendo a la línea de comandos tantos modificadores /r: como librerías necesitásemos. Veremos ejemplos de la utilización del compilador en línea más adelante.

A lo largo de los siguientes artículos, iremos profundizando en las distintas características de uso del lenguaje a través de ejemplos. De esta forma, al analizar una simple aplicación vamos a ir revisando el lenguaje en sus características más importantes.

Anexo 1: Accesibilidad de los tipos y métodos de una clase

De cara al futuro, añadimos a continuación las tablas que definen, respectivamente, los calificadores de ámbito,la accesibilidad y los modificadores de acceso, a modo de referencia. Como el lector tendrá oportunidad de comprobar, las palabras reservadas que se indican en las tablas adjuntas, aparecen constantemente en los programas escritos en C#. Por tanto, y resumiendo la situación podemos desglosarla de la siguiente forma:

Calificadores de ámbito de accesibilidad para tipos y métodos de una clase		
Declaracion	**Nivel de Accesibilidad**	**Tipo de accesibilidad**
Public	Sin restricciones	Declara miembros accesibles desde cualquier otra clase
Protected	Limitado a la clase contenedora y tipos derivados	Declara miembros accesibles únicamente desde dentro de la clase y sus clases derivadas
Internal	Limitado al proyecto actual	Declara miembros accesibles únicamente desde dentro de mismo módulo de ensamblaje (*assembly*)
Protected Internal	Limitado al proyecto actual o tipos derivados	
Private	Limitado al tipo contenedor.	Declara miembros accesibles únicamente desde la clase o estructura en la que son declarados.

Tabla 1. Calificadores de ámbito de accesibilidad

La accesibilidad de un tipo o método se especifica en C# mediante uno de los siguientes modificadores (**modifiers**) o declaraciones, tal y como aparecen en la tabla adjunta:

De las declaraciones indicadas en la tabla adjunta la única combinación permitida es ***protected internal***. Por otro lado los Espacios Calificados no están sujetos a ningún tipo de restricción. Además, el contexto de la declaración, impone restricciones adicionales, y en caso de no utilizar ningún modificador, se asume la declaración por defecto para el contexto de uso que corresponda.

Los miembros de…	Disponen de una accesibilidad por defecto del tipo:	Y a sus miembros, se les permite un nivel de declaraciones de acceso:
enum	public	**Ninguno**
class	private	public protected internal private protected internal
interface	public	**Ninguno**
struct	private	public internal private

Tabla 2. Accesibilidad

Entendemos por *dominio de accesibilidad*, aquellas secciones del código de un programa en las cuales se puede hacer referencia a un miembro determinado. Según esto, la accesibilidad de un tipo anidado depende de su ***dominio de accesibilidad***, que viene determinado por dos aspectos: *el nivel de accesibilidad del miembro* y el *dominio de accesibilidad del tipo contenedor*. Sin embargo, el dominio de accesibilidad de un tipo anidado

[1] no puede exceder el del tipo contenedor.

Otros Modificadores

La lista completa de modificadores en C# es la siguiente:

Modificador	Propósito
Modificadores de acceso (ya comentados): **public**, **private**, **internal** y **protected**	**Especifican declaraciones de accesibilidad de tipos y tipos miembro.**
abstract	Indica que una clase sirve **solamente** de clase base o clase abstracta para otras clases. No se pueden instanciar

	objetos a partir de ella.
const	Especifica valores para campos o variables que no pueden modificarse.
event	Declara un evento.
extern	Indica un método que se implementa de forma externa. (Por ejemplo, declara una función de librería de una API).
override	Indica que se trata de una **nueva** implementación de un miembro virtual heredado de una clase base.
readonly	Declara un campo cuyo valor sólo puede asignarse una vez, bien como parte de la declaración, o en un constructor de esa misma clase..
sealed	Especifica una clase de la que no puede heredarse. (Pero sí instanciar objetos a partir de ella)
static	Declara un miembro asociado con el tipo mismo (la clase, estructura, etc), en lugar de pertenecer a un objeto concreto. Interviene en el concepto de clases globales, comentado más arriba.
virtual	Declara que el método que sigue se podrá sobrescribir en una clase derivada de ella.

Tabla 3. Modificadores de Acceso

Siguiendo con nuestro razonamiento, vemos que el siguiente modificador de nuestra declaración del método *Main* era ***static***. Según la tabla adjunta, eso significa que puede ser llamado desde otro proyecto, o directamente, por CLR, al iniciarse la aplicación.

---

[1] Un tipo anidado (*nested type*) es aquel que se declara dentro del ámbito de otro tipo y suele ser útil para encapsular detalles de la implementación de éste, tales como enumeraciones, etc.

El Common Type System (Sistema Común de Tipos de Datos)

Una vez hecha la introducción básica al lenguaje C#, es costumbre continuar con la descripción de los elementos fundamentales del lenguaje, comenzando por los tipos de datos que es capaz de manejar. Y una de las características principales que permite la integración de los diferentes lenguajes soportados por el entorno común de ejecución (CLR), se basa en la existencia un conjunto común de tipos de datos que coinciden para cualquier lenguaje. No importa que estemos utilizando VB.NET, C# o -el recientemente anunciado por Fujitsu- COBOL.NET, los tipos de datos definidos son siempre son los mismos, y su definición e implantación se agrupan bajo el epígrafe de ***Sistema Común de Tipos de Datos*** ó **Common Type System** (desde ahora, **CTS**).

La principal ventaja que tiene un sistema de tipos común radica en la seguridad. Seguridad en la conversión de tipos y garantía de que cualquier lenguaje que utilizará el **CTS** podrá interpretar e interactuar con módulos creados en otros lenguajes sin *traducciones*. Además, ciertas operaciones no están permitidas de forma explícita, como algunas conversiones, para las que es preciso especificarlo de forma explícita. Piense el lector, lo que sucedía, por ejemplo en **Visual Basic**: en las versiones anteriores, podíamos llamar a funciones del API de **Windows**, pero como las definiciones de esas funciones habían sido originalmente hechas en **C++**, muchas veces se requerían adaptar los tipos de datos **VB** a los de **C++**, para que la llamada a la función no terminase en un error de ejecución. Además, esta situación facilita la construcción de componentes, que -más incluso que los ejecutables- necesitan interactuar con cualquier código.

El siguiente gráfico, muestra esta situación, esquematizada según una división conceptual que se basa en la ubicación de los tipos de datos: la pila (***stack***), o el montón (***heap***):

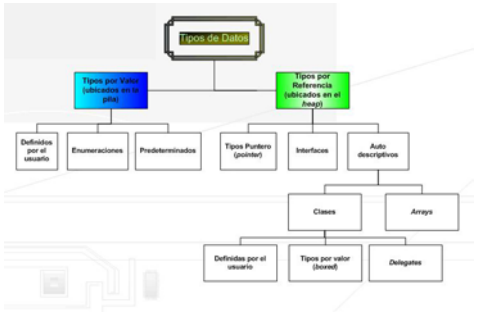

Figura 1. Esquema general del Sistema Común de Tipos (CTS)

Analizaremos con más detalle este diagrama para explicar brevemente su funcionamiento. Como anticipábamos, la primera división importante, se establece a la hora de distinguir la ubicación de los tipos de datos (*stack* ó *heap*).

Diferenciación entre la pila (ó *stack)* y el montón (*heap*)

La pila es un área de memoria utilizada para el almacenamiento y recuperación de valores de longitud fija. Hay que recordar que cada programa en ejecución (cada proceso, en términos de Windows), mantiene su propia pila, a la que ningún otro programa puede acceder. Con eso se garantiza la seguridad. Cuando se llama a función todas las variables locales a esa función son almacenadas (instrucciones

PUSH de **Ensamblador**) en la pila, para ser posteriormente recuperadas (instrucciones **POP**) cuando el flujo de ejecución sale de la función. El típico error "**stack overflow**" se produce precisamente cuando una rutina (por ejemplo, alguna de tipo recursivo) almacena demasiadas variables en la pila y ésta se desborda.

La otra ubicación posible para los datos de un programa es el montón (o *heap*), que se usaba en **C/C++** para almacenar valores de longitud variable (por ejemplo arrays de chars), o datos cuyo ciclo vital superaba el de una función de manera que se puede hacer referencia a ellos fuera de la función que los define. Aunque la situación en **C#** es similar, la diferencia con **C/C++** es que los datos de este tipo (objetos), se almacenan en una zona denominada ***managed heap***, algo así como ***montón administrado***, de cuya gestión se encarga el **Entorno Común de Ejecución** (**CLR**). Internamente, opera de forma más eficiente, y la elección entre el uso de la pila o el montón la toma el compilador, dependiendo del tipo de dato.

Declaraciones de tipos por valor

Esto significa que cada instrucción es separada por el compilador y se asocian los nombres X e Y a sendas direcciones de memoria, almacenando copias independientes de los datos, como vemos en la figura adjunta:

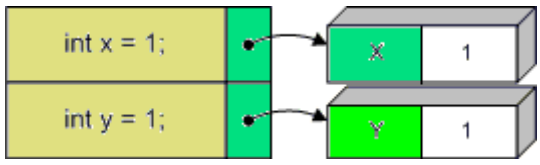

Figura 2. Asignación de tipos predeterminados en la pila

Aparte de tener una incidencia en el rendimiento (el acceso a la pila es más rápido que el acceso al montón, por razones que tienen que ver con los registros de la CPU, pero que no vamos a detallar aquí), en el primer caso, estamos hablando de tipos que no tienen una **identidad** establecida. Se trata de valores, de secuencias de bits situadas en memoria. El sistema se las *arregla* para hacer corresponder los nombres de las variables con las direcciones de memoria en las que estas se ubican, pero cada variable guarda una copia distinta del valor, aunque este sea el mismo (como en la figura 2).

Sin embargo, en el caso de los tipos por referencia, se habla de una combinación de **identidad**, **ubicación** y **valores** (secuencias de bits). De acuerdo con esto, dos objetos se consideran iguales NO SI ALMACENAN EL MISMO VALOR, SINO CUANDO APUNTAN A LA MISMA DIRECCIÓN DE MEMORIA (que, como consecuencia, va a implicar que los valores sean los mismos, pero que supone que *el cambio de estado en una de las variables de objeto va a significar también el cambio en todas las otras variables que hagan referencia al mismo objeto* (o sea, que apunten a la misma dirección).

La diferencia queda esquematizada en la figura 3, adjunta:

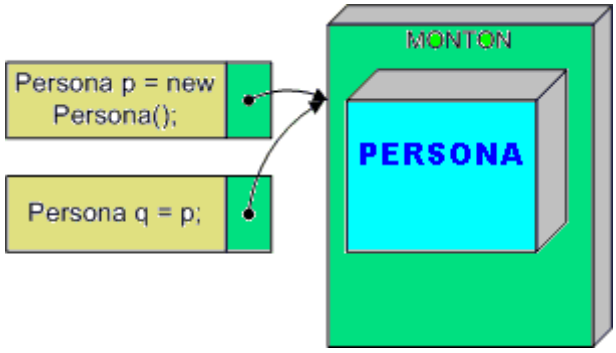

Figura 3. Asignación de variables de objeto que apuntan a la misma dirección de memoria

Nota*: Esto significa que aunque se declaren variables miembro de una clase (campos) utilizando las declaraciones abreviadas (int, double, etc), como son pertenecientes a un objeto situado en el montón, ellas, a su vez, estarán también situadas en el montón. La ubicación depende de la declaración inicial y del ámbito.*

Boxing y Unboxing

Como en ocasiones es necesario utilizar un tipo por valor como si fuera un objeto y viceversa, en C# se denominan boxing y unboxing a los procesos de conversión que permiten este tratamiento. En el primero de los procesos, no se necesita ningún tipo de casting mientras que en el contrario sí lo es (en principio, podríamos convertirlo en cualquier tipo de dato).

Proceso de Boxing y Unboxing

```
  object x = y;  //Proceso de boxing: y se convierte a referencia para
permitir la asignación

  int z = (int) x; //Proceso de Unboxing: x se convierte a entero para
```

Existe un caso especial que tendremos ocasión de probar más adelante y es el de los tipos de datos **string**, que realmente se almacenan en el montón, pero de cara al usuario y para la facilidad de manejo de pueden manejar como si se ubicaran en la pila.

Relación de tipos predeterminados en .NET

El entorno .NET provee de una serie de tipos predeterminados que permiten la declaración de variables y su asignación directa sin necesidad de instanciar ningún objeto. Se les denomina ***tipos predeterminados*** y un ejemplo de ellos es ***int***. Así, podemos usar estos *atajos* para simplificar el código. Por ejemplo, las dos declaraciones siguientes, aunque similares (puesto que manejan valores del mismo tipo), se declaran de forma distinta y existen algunas distinciones en su utilización:

Tipos predeterminados en .NET						
Categoría	**Nombre de Clase**	**Descripción**	**Tipo de datos Visual Basic**	**Tipo de datos C#**	**Tipo de datos Managed Extensions de C++**	**Tipo de datos JavaScript**
Enteros	Byte	entero sin signo de 8-bits	**Byte**	**byte**	**char**	**byte**
	SByte	entero con signo de 8-bits No cumple con CLS	**SByte** No predeterminado	**sbyte**	**signed char**	**SByte**
	Int16	entero con signo de 16-bits	**Short**	**short**	**short**	**short**
	Int32	entero con signo de 32-bits	**Integer**	**int**	**int** o **long**	**int**
	Int64	entero con signo de 64-bits.	**Long**	**long**	**__int64**	**long**
	UInt16	entero sin signo de 16-bits. No cumple con CLS.	**UInt16** No predeterminado	**ushort**	**unsigned short**	**UInt16**
	UInt32	entero sin signo de 32-bits. No cumple con CLS.	**UInt32** No predeterminado	**uint**	**unsigned int** -or- **unsigned long**	**UInt32**
	UInt64	entero sin signo de 64-bits No cumple con CLS.	**UInt64** No predeterminado	**ulong**	**unsigned __int64**	**UInt64**
Coma Flotante	Single	Un número en coma flotante de precisión simple (32-bits)	**Single**	**float**	**float**	**float**
	Double	Un número en coma flotante de doble precisión (64-bits)	**Double**	**double**	**double**	**double**
Lógicos	Boolean	Un valor Boolean (**true** o **false**).	**Boolean**	**bool**	**bool**	**bool**
Otros	Char	Un carácter Unicode (16-bits) .	**Char**	**char**	**wchar_t**	**char**
	Decimal	Un valor decimal de 96-bits.	**Decimal**	**decimal**	**Decimal**	**Decimal**

	IntPtr	entero con signo cuyo tamaño depende de la plataforma subyacente (valor 32- bits en plataformas de 32-bits valor 64-bits en plataformas de 64-bits..	**IntPtr** No predeterminado	**IntPtr** No predeterminado.	**IntPtr** No predeterminado.	**IntPtr**
	UIntPtr	entero sin signo cuyo tamaño depende de la plataforma subyacente (valor 32- bits en plataformas de 32-bits valor 64- bits en plataformas de 64-bits. No cumple con CLS.	**UIntPtr** No predeterminado	**UIntPtr** No predeterminado	**UIntPtr** No predeterminado	**UIntPtr**
objetos	Object	El objeto raíz de la jerarquía	**Object**	**object**	**Object***	**Object**
	String	Cadena inmutable de caracteres Unicode y longitud fija.	**String**	**string**	**String***	**String**

Nótese que el tipo de datos **string**, aparece en el apartado de objetos (*Class Objects*), y sin embargo en su utilización directa no hay diferencia respecto a los tipos estándar predeterminados que aparecen al comienzo de la tabla. De tal forma, podemos asignar valores a variables de tipo cadena utilizando una sintaxis muy similar a la utilizada con los enteros. Esto es radicalmente distinto a la utilización que se hacia de las cadenas en C/C++ mediante arrays de *chars*.

Así, operaciones tediosas en esos lenguajes, como la concatenación de cadenas, o el análisis de caracteres dentro de las mismas es mucho más simple. Por ejemplo, la concatenación se consigue simplemente mediante el siguiente código:

Asignación y uso de tipos *string* en una concatenación

```
string cad2 = "Y esto es otra";
```

No obstante, seguimos manejando referencias de forma interna (internamente una instancia de **System.Object**, se ubica en el montón). Eso significa que la primera vez que asignamos una cadena a otra, los valores apuntan a la misma ubicación (o secuencia de caracteres), pero cuando la segunda variable es reasignada a una constante de caracteres, o una expresión que pueda evaluarse como tal, se crea

una copia independiente en la memoria, de forma que cada variable mantenga su conjunto distinto de datos, como podemos ver en el siguiente ejemplo:

Asignación y uso de tipos *string* en una concatenación

```
class EjemploCadena

{

   public static void Main()

   {

     string s1 = "Una cadena";

     string s2 = s1;

     Console.WriteLine("Cadena 1: {0}", s1); //Imprime "Una cadena"

     Console.WriteLine("Cadena 2: {0}", s2); //Imprime "Una cadena"

     s1 = "Otra cadena";

     Console.WriteLine("Cadena 1: {0}", s1); //Imprime "Otra cadena"

     Console.WriteLine("Cadena 2: {0}", s2); //Imprime "Una cadena"

   }
```

Como podemos ver, la primera vez, los valores son idénticos, pero una vez que se vuelve a asignar el valor cada referencia mantiene datos propios. Los literales de cadena en C# se deben encerrar siempre entre comillas dobles (lo contrario tendería a ser interpretado como un dato **char**, produciendo un error de compilación). El lenguaje define una serie de caracteres que sirven para el formato de cadenas.

Los elementos del lenguaje

Se decía que los tres componentes funcionales más importantes de una clase eran, sus campos (que definen su **estado**), sus métodos (que definen su **funcionalidad**) y sus eventos (que establecen la forma de **comunicación** con el exterior de la clase). Estos elementos, junto a las palabras reservadas del lenguaje que sirven de modificadores, permiten implementar en una clase las 3 características de la OOP (Polimorfismo, Encapsulación y Herencia), enriquecidas al evolucionar los lenguajes, como sucede en C#, mediante aportaciones nuevas, como los indizadores, las propiedades, los atributos o los delegados, y también nuevas o modificadas formas de tratar elementos conocidos. Vamos a continuar este repaso básico del lenguaje hablando de los operadores.

Operadores

Entendemos por operador un símbolo que indica que debe realizarse una operación sobre uno o más argumentos, que reciben el nombr de operandos. A los operadores que actúan sobre un único argumento, se les llama operadores *unarios.*

Categoría del Operador	Operadores
Primarios	(x), x.y, f(x), a[x], x++, x--, new, typeof, sizeof, checked, unchecked
Unarios	+, -, !, ~, ++x, --x, (T)x
Multiplicativos	*, /, %
Aditivos	+, -
De desplazamiento	<<, >>
Relacionales	<, >, <=, >=, is
Igualdad	= =
AND Lógico	&
XOR Lógico	^
OR Lógico	|
AND Condicional	&&
OR Condicional	||
Condicional	?:
Asignación	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Tabla 1: Lista de operadores en C# por orden de precedencia.

Como podemos ver, se dispone de un amplio conjunto de operadores basado en los existentes para los lenguajes C++/Java. Aunque la mayor parte de ellos siguen las reglas establecidas para otros lenguajes, conviene establecer algunas de las reglas generales más importantes que se cumplen en su tratamiento e implantación, que son las siguientes:

1. La precedencia de los operadores es la que se indica en la tabla adjunta.
2. Todos los operadores, excepto el de asignación (=), se evalúan de izquierda a derecha. Esto significa que se da propiedad transitiva en los operadores. O sea, que la expresión
   A+B+C es equivalente a (A+B)+C. Como siempre, se utilizan los paréntesis para establecer precedencia específica en cualquier expresión. Nótese, además, que en C#, se utilizan distintos símbolos para el operador de

asignación (=) y el de comparación (= =), evitando errores de interpretación, como sucede en otros lenguajes, al estilo de Visual Basic.

3. El operador (**.**) se utiliza para especificar un miembro de una clase o estructura. El operando de la izquierda simboliza la clase y el de la derecha el miembro de esa clase. Se eliminan otros operadores presentes en el lenguaje C++, tales como (**->**,**::)**, para simplificar la sintaxis.
4. El operador ([]) se utiliza para hacer referencia a los elementos de un array. Todos los arrays en .NET comienzan en 0.
5. Los operadores (++ y --) se utilizan al estilo de C++, como operadores de autoincremento y autodecremento. Pueden ser utilizados dentro de expresiones.
6. ***new*** es el operador usado para invocar métodos constructores a partir de definiciones de clases.
7. ***typeof*** es el operador utilizado mediante la librería ***Reflection*** para obtener información acerca de un tipo.
8. ***sizeof*** se utiliza para determinar el tamaño de un tipo estándar expresado en bytes. Solo puede ser utilizado con los llamados ***value types*** (tipos por valor), o miembros de clases, pero no con las propias clases. Además, sólo pueden usarse dentro de bloques ***unsafe*** (código C++ embebido en un bloque ***unsafe*** de C#).
9. ***checked*** y ***unchecked*** se utilizan para la gestión de errores, dentro de estructuras ***try-catch-finally***. Lo veremos con más detalle, en el apartado de tratamiento de errores.
10. Los operadores de asignación compuesta se utilizan para simplificar expresiones en las que un operando aparece en los dos miembros de la asignación. Por ejemplo, podemos simplificar un contador expresándolo como x += 1, en lugar de x = x + 1.

Comentarios

Los comentarios en C# son uno de los aspectos extendidos del lenguaje, respecto a sus predecesores. Aunque los comentarios en línea se hacen mediante la doble barra vertical (**//**), de manera similar a C++: Y los bloques de comentario, mediante la sintaxis consistente en encerrar todo el comentario entre los símbolos (**/*** y ***/**):

Comentarios en C#

```
//Esto es un comentario
System.Console.Readline() //y esto, otro
/* Esto es un comentario
según la segunda sintaxis */
```

No obstante, se han añadido posibilidades adicionales de cara a la documentación del código. Según esto, el propio IDE de Visual Studio, utiliza tres barras verticales (///), para incluir comentarios que pueden más tarde usarse en la confección de un fichero de texto en sintaxis XML:

Comentarios en C# para documentación XML

```
///<summary>
///Resumen de la descripción de la clase Test
///</summary>
```

Observe el lector que la información añadida se incluye dentro de dos etiquetas XML de nombre <summary> (sumario). Posteriormente, una opción de compilación (/**doc:<Fichero XML>**), permite que se genere un fichero adicional con dicha

extensión que puede ser visualizado mediante un navegador, u otro programa adecuado. En el caso de que deseemos modificar las plantillas incluidas en el IDE para la construcción de módulos de código, se pueden modificar dichas etiquetas para que tengan el sentido y la estructura que nos interesen, y posteriormente, mediante ficheros de **Hojas de Estilo Extendidas (XSL/T)**, generar directamente formatos HTML, que puedan convertirse en Ayudas, mediante el editor de Ayudas que también se incluye en la *suite* de Visual Studio.

Observamos aquí algo que se convierte en una constante de todo el entorno de desarrollo: XML es el estándar actual para la escritura de ficheros de datos, y es el formato que se usa siempre que es posible (veremos esto mismo asociado a los ficheros de configuración externa de una aplicación, a los datos devueltos por un servidor mediante ADO.NET, y en muchas otras situaciones).

Identificadores

Un identificador es una palabra definida por el programador, y distinta de cualquier palabra reservada del lenguaje, que se usa para especificar un elemento único dentro de un programa (clase, variable, método, constante, atributo, etc.).

Las reglas de los identificadores explicadas en esta sección se corresponden exactamente con las recomendadas por la norma **Unicode 3.0**, Informe técnico 15, Anexo 7, con excepción del carácter de subrayado, que está permitido como carácter inicial (como era tradicional en el lenguaje de programación C), los caracteres de escape Unicode están permitidos en los identificadores y el carácter "@" está permitido como ***prefijo*** para habilitar el uso de palabras clave como identificadores, siempre y cuando no exista otra declaración idéntica sin dicho prefijo (sería interpretado como una declaración duplicada). Además, el carácter @, tiene una utilización especial como prefijo de cadenas de caracteres, cuando lo que se quiere indicar es que la cadena posee delimitadores que deben ser interpretados como literales, como es el caso de la barra separadora de directorios: \).

Literales

Entendemos por literal a un conjunto de caracteres alfanuméricos que representan los valores que puede adoptar un tipo básico del lenguaje. Según esto, existen tantas clases de literales como de tipos en el Common Type System: lógicos, enteros, reales, de caracteres, de cadena, y el literal nulo.

Algunos ejemplos serían:

Literales lógicos: Representan a los valores lógicos cierto y falso:

true, **false** (con minúsculas)

Literales enteros: Representan números enteros en notación decimal o hexadecimal:

0, -3276, 0x1A

Literales reales: Representan números en formato decimal.

0.0, -1000.45, 1.23E-9 (notación científica)

Literales de caracteres: Representan un carácter individual, típicamente encerrado entre comillas simples: 'a'. También se utilizan para la representación de "secuencias de escape" (representaciones alternativas de un carácter basadas en su posición en una tabla de definiciones estándar: ASCII, Unicode, etc.).

En C# y en las Extensiones administradas de C++, el carácter de barra invertida hace que el siguiente carácter de la cadena de formato se interprete como una secuencia de escape. Se utiliza con las secuencias tradicionales para de aplicación de formato, como "\n" (nueva línea).

Una secuencia de escape de caracteres Unicode representa un carácter Unicode. Es importante tener en cuenta que Las secuencias de escape de caracteres Unicode se procesan en identificadores, literales de caracteres y literales de cadenas regulares. Un carácter de escape Unicode no se procesa en ninguna otra ubicación (por ejemplo, para formar un operador, un elemento de puntuación o una palabra clave).

La tabla adjunta lista **las secuencias de escape Unicode**:

Secuencia de escape	Nombre del carácter	Codificación Unicode
\'	Comilla simple	0x0027
\"	Comilla doble	0x0022
\\	Barra invertida	0x005C
\0	Null	0x0000
\a	Alerta	0x0007
\b	Retroceso	0x0008
\f	Avance de página	0x000C
\n	Nueva línea	0x000A
\r	Retorno de carro	0x000D
\t	Tabulación horizontal	0x0009
\v	Tabulación vertical	0x000B

Tabla 2: secuencias de escape Unicode.

Literales de cadena: (Siempre entre comillas dobles")

"Cadena", "C:\Windows"

El Literal nulo: En C# se representa mediante la palabra reservada **null**. Se usa como valor de las variables tipo **Object** no inicializadas (que contienen una referencia nula).

Atributos

A pesar de que C# es un lenguaje imperativo, sigue las normas de los lenguajes de esta categoría, y contiene elementos declarativos. Ya hemos visto que la accesibilidad de un método de una clase se especifica mediante su declaración como **public, protected, internal, protected internal** o **private**.

Al objeto de que los programadores pueden definir sus propias clases de información declarativa, C# generaliza esta capacidad, permitiendo anexar dicha información a entidades del programa y recuperarla en tiempo de ejecución. Tal información declarativa adicional se realiza mediante la definición y el uso de atributos.

Más adelante, el programa puede recuperar dicha información mediante la característica **Reflection** que consiste en la capacidad de un programa de utilizar una librería de clase base para averiguar lo que necesite acerca de sus propios Metadatos. Analizaremos más en profundidad esta característica en artículos posteriores.

También se utilizan los atributos para definir acciones que se deben de llevar a cabo en tiempo de ejecución, como sucede, por ejemplo, en la técnica denominada **Platform Invoke,** para declarar una función perteneciente a una librería del API de Windows, como si fuera propia y poder utilizarla. El siguiente ejemplo es una muestra de ésta característica:

Uso del atributo DLLImport en la declaración de una función del API de Windows

```
[DllImport("version.dll")]

   public static extern bool GetFileVersionInfo (string sFileName,

      int handle, int size, byte[] infoBuffer);
```

Es notable recalcar, que Platform Invoke es un mecanismo independiente de la plataforma, y por tanto "*es el recomendado para las llamadas a funciones propias del sistema cuando se trabaja en plataformas no-Windows*", tal y como nos recuerda el director del proyecto **Mono** (de la iniciativa de software abierto **Open Source), Miguel de Icaza,** encargado de la conversión de la plataforma .NET y el compilador de C#, para Linux, en el artículo publicado al respecto en el primer número de la revista ***Dr.Dobbs*** en castellano.

Formatos de salida

En C#, se incluye una variada gama de formatos posibles para la representación de valores numéricos y alfanuméricos. Al tratarse este de un problema muy típico para los programadores, hemos querido hacer especial hincapié en este punto, y detallar en un conjunto de tablas todas estas posibilidades. Podemos dividir esta característica en dos grandes grupos, el de formatos numéricos y el de formatos de cadenas.

Cadenas de formato numérico personalizado

Los caracteres que se pueden utilizar para crear cadenas de formato numérico personalizado y sus definiciones, se muestran en la tabla siguiente. Téngase en cuenta que la Configuración Regional del Panel de control influye en algunos de los modelos generados por estos caracteres. Los equipos que emplean referencias culturales diferentes mostrarán modelos diferentes. La tabla adjunta, tomada de la documentación oficial del producto, ilustra todas estas posibilidades.

Tabla para crear cadenas de formato numérico personalizado		
Carácter de formato	**Nombre**	**Descripción**
0	Marcador de posición cero	Si el valor al que se está dando formato tiene un dígito en la posición donde aparece el '0' en la cadena de formato, entonces ese dígito se copia en la cadena de salida. La posición del '0' que aparece más a la izquierda antes del separador decimal y la del '0' que está más a la derecha después del separador decimal determinan el intervalo de dígitos que están siempre presentes en la cadena de salida.
#	Marcador de posición de dígito.	Si el valor al que se está dando formato tiene un dígito en la posición donde aparece '#' en la cadena de formato, entonces ese dígito se copia en la cadena de salida. En caso contrario, no se almacena nada en esa posición de la cadena de salida. Hay que advertir que este indicador nunca muestra el carácter '0' si éste no es un dígito significativo, incluso si '0' es el único dígito de la cadena. Sólo mostrará el carácter '0' si es un dígito significativo del número que se muestra.
.	Separador decimal	El primer carácter '.' de la cadena de formato determina la ubicación del separador decimal en el valor con formato y se omite cualquier carácter '.' adicional. El carácter real que se utiliza como separador decimal viene determinado por la propiedad *NumberDecimalSeparator* del objeto *NumberFormatInfo* que controla la aplicación de formato.
,	Separador de miles y escala numérica	El carácter ',' tiene una doble utilidad. En primer lugar, si la cadena de formato contiene un carácter ',' entre dos marcadores de posición de dígitos (0 ó #) y a la izquierda del separador decimal, si hay, entonces el resultado tendrá separadores de miles insertados entre cada grupo de tres dígitos a la izquierda del separador decimal. El carácter real que se utiliza como separador decimal en la cadena de salida viene determinado por la

		propiedad NumberGroupSeparator del objeto NumberFormatInfo actual que controla la aplicación de formato. En segundo lugar, si la cadena de formato contiene uno o más caracteres ',' situados inmediatamente a la izquierda del separador decimal, entonces el número se dividirá entre el número de caracteres ',' multiplicado por mil antes de que se le dé formato. Por ejemplo, la cadena de formato '0,,' presentará una cantidad de 100 millones como 100. El uso del carácter ',' para indicar la escala no incluye los separadores de miles en el número al que se ha dado formato. Así pues, para indicar la escala de un número para un millón e insertar separadores de miles, hay que utilizar la cadena de formato '#,##0,,'.
%	Marcador de posición de porcentaje.	La presencia de un carácter '%' en una cadena de formato hace que se multiplique un número por 100 antes de que se le dé formato. El símbolo adecuado se inserta en el número en la posición en que aparece '%' en la cadena de formato. El carácter de porcentaje que se utiliza depende de la clase NumberFormatInfo actual.
E0 E+0 E-0 e0 e+0 e-0	Notación científica	Si alguna de las cadenas 'E', 'E+', 'E-', 'e', 'e+' o 'e-' aparece en la cadena de formato y lleva a continuación por lo menos un carácter '0', entonces se le da formato al número mediante notación científica con una 'E' o una 'e' insertadas entre el número y el exponente. El número de caracteres '0' que sigue al indicador de notación científica determina el número mínimo de dígitos para el exponente. Los formatos 'E+' y 'e+' indican que un carácter de signo (más o menos) debe preceder siempre al exponente. Los formatos 'E', 'E-', 'e' o 'e-' indican que un carácter de signo sólo debe preceder a exponentes negativos.
\	Carácter de escape	En C# y en las Extensiones administradas de C++, el carácter de barra invertida hace que el siguiente carácter de la cadena de formato se interprete como una secuencia de escape. Se utiliza con las secuencias tradicionales para de aplicación de formato, como "\n" (nueva línea). En algunos lenguajes, el carácter de escape debe ir precedido por otro carácter de escape cuando se utiliza como un literal. En caso contrario, el compilador interpreta el carácter como una secuencia de escape. Hay que emplear la cadena "\\" para mostrar "\". Hay que tener en cuenta que Visual Basic no admite el carácter de escape; sin embargo ControlChars proporciona la misma funcionalidad.
'ABC' "ABC"	Cadena literal	Los caracteres situados entre comillas sencillas o dobles se copian literalmente en la cadena de salida y no afectan al proceso de dar formato.
;	Separador de secciones	El carácter ';' se utiliza para separar secciones para los números positivos, negativos y cero de la cadena de formato.

Otros	Todos los demás caracteres	Todos los demás caracteres se copian en la cadena de salida como literales en la posición en la que aparecen.

Tabla 1: cadenas de formato

Un ejemplo de funcionamiento sería como sigue:

Declaraciones por valor

```
class Class1
{
static void Main()
{
Console.Write("{0:C} \n", 2,5);
Console.Write("{0:D5} \n", 25);
Console.Write("{0:E} \n", 250000);
Console.Write("{0:F2} \n", 25);
Console.Write("{0:F0} \n", 25);
Console.Write("{0:G} \n", 2,5);
Console.Write("{0:N} \n", 2500000);
Console.Write("{0:X} \n", 250);
Console.Write("{0:X} \n", 0xffff);
Console.ReadLine();
}
}

//La salida por pantalla es:

2,00 ?
00025
2,500000E+005
25,00
25
2
2.500.000,00
FA
```

---

system.object: un gran tipo

La raíz de todos los tipos de datos en C#, es, como ya hemos apuntado, System.Object. Esto, garantiza que cualquier tipo del CTS comparte un conjunto de características común. En concreto, existen 4 métodos públicos asociados con el tipo base, que son heredados por todos los objetos, tal y como se muestran en la tabla siguiente:

Métodos públicos de System.Object

bool Equals()	Compara dos referencias a objetos en tiempo de ejecución para comprobar si apuntan al mismo objeto. Si es así el método devuelve *true*. En los tipos por valor, se devuelve *true* si los datos almacenados son idénticos.
int GetHashCode()	Obtiene el código *hash* de un objeto. Las funciones Hash se usan cuando se desea situar dicho código en una tabla *hash* por razones de rendimiento.
Type GetType()	Se utiliza con métodos del *namespace reflection* para obtener la información del tipo de un objeto dado.
string ToString	Por defecto, se utiliza para obtener el nombre del objeto. Admite reemplazo (overload) por clases derivadas para devolver cadenas de representación más adecuadas a las necesidades del usuario.

Además de estos métodos públicos, *System.Object* posee dos métodos protegidos (*protected*): ***Finalize***() que es el equivalente al método destructor de otros lenguajes, pero teniendo en cuenta que esa no es la forma que se utiliza en .NET, y *MemberwiseClone()* que devuelve una copia de un objeto ignorando todas las referencias a que ese objeto pudiera apuntar. A este tipo de copia se la llama *Copia superficial* del objeto.

Conversión de tipos

Una de las características fundamentales del tratamiento de tipos de datos es la conversión entre ellos. En C++, esta operación recibe el nombre de *casting.* Una forma conveniente de abordar el problema es distinguir entre la **conversión implícita** (llevada a cabo por el compilador) y la **conversión explicita**, que debe indicarse de forma expresa en el código fuente. En C#, se sobreentiende que el conjunto de conversiones explícitas incluye al de conversiones implícitas, más las siguientes:

- ***Conversiones explícitas numéricas***: Son aquellas en las que se indica en el código fuente el tipo de dato a convertir mediante la técnica del *casting*, hacia otro tipo numérico, como veremos a continuación.
- ***Conversiones explícitas de enumeración***: De ***sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double*** o ***decimal*** a cualquier tipo ***enum*** así como su opuestos **(**de ***enum*** *a los tipos citados),* más todos aquellos de cualquier tipo ***enum*** a cualquier otro tipo ***enum***.
- ***Conversiones explícitas de referencia***: en general, aquellas en las que están implicados tipos por referencia (ver Tabla 1).
- ***Conversiones explícitas de interfaz***: similares a las anteriores, pero en las que uno de los operandos se refiere a una interfaz, en lugar de a una clase.
- ***Conversiones Unboxing***. (de las que ya hemos visto un ejemplo)
- ***Conversiones explícitas definidas por el usuario***: Si el lector ha revisado la lista de palabras reservadas del lenguaje, probablemente, haya reparado en la presencia de dos que tienen que ver directamente con el tema que nos ocupa: ***implicit*** y ***explicit***. Estas definiciones pueden usarse en el código para establecer la forma exacta en que deseamos que se lleve a cabo una conversión de un tipo definido por el programador.

La conversión implícita sólo es posible si el tipo convertido es mayor que el original. Veamos en un ejemplo esta situación:

Conversiones implicitas y explícitas

```
long  x = 0;
int y = (int) x;
/* Si no lo indicáramos expresamente, el compilador daría un error del
```

```
tipo: “no se puede convertir implícitamente el //tipo long a int” */
short z = (short) x;
object z = (object) x; //Aquí, el moldeado permite llamar al
método ToString() sobre el objeto z
System.Console.WriteLine(“{0}”, z.ToString() + “-convertido”);
Salida:
1-convertido
```

En C#, la palabras reservadas *checked* (conversión comprobada) y *unchecked* (conversión no comprobada) pueden usarse para garantizar que la conversión de un tipo ha sido correcta, lo que requiere una expresión extra entre paréntesis (moldeado múltiple), o bien para ignorar explícitamente la excepción que se produciría, por ejemplo, en un caso de desbordamiento. Veamos el siguiente ejemplo:

Conversiones checked y unchecked

```
int y = checked ((int)x); // El valor de y es de 1234567. (Se pierden
los decimales)
```

Otros casos de moldeado múltiple son comunes al trabajar con la jerarquía de clases de .NET cuando se quieren convertir tipos definidos por el usuario en otros o se utiliza conversión de interfaces. La regla, en este caso, es que siempre existe una conversión implícita entre un tipo complejo y su tipo base.

El trabajo con clases en C#

El lector conoce, sin duda los fundamentos de la programación orientada a objetos, por lo que no creo necesario abundar aquí sobre el tema. Sí parece adecuado, no obstante, abordar la construcción y estudio de las clases siguiendo tales principios fundamentales, que podríamos dividir conceptualmente en dos puntos de vista: el estructural y el funcional.

Entendemos, en el primer caso, aquellos elementos que dan a la clase una arquitectura, y que tradicionalmente, se han subdividido en: campos, métodos, operadores y eventos. En el caso de C#, será necesario además, incluir algunos elementos nuevos que constituyen precisamente las aportaciones más importantes del lenguaje, como son las propiedades (**properties**), los indizadores (**indexers**) y los delegados (**delegates**). Se trata de una división meramente conceptual, ya que estos elementos se traducen a su vez en características del funcionamiento de una clase, y por tanto aportan siempre un componente funcional, a la vez que estructural.

Campos

Un campo es una variable miembro de una clase, que sirve para almacenar un valor. Admiten varios modificadores, que definen el comportamiento del campo en tiempo de ejecución: **private, public, protected, internal, static, readonly** y **const**. A los cuatro primeros, se les denomina modificadores de acceso. Comentamos a continuación algunos de sus significados.

La tabla siguiente recoge los niveles de accesibilidad establecidos en C#, junto con su significado:

Accesibilidad declarada	Significado
public	Acceso no restringido
protected	Acceso limitado a la clase contenedora o a los tipos derivados de esta clase.
internal	Acceso limitado al proyecto actual.
protected internal	Acceso limitado al proyecto actual o a los tipos derivados de la clase contenedora.
private	Acceso limitado al tipo contenedor.

Private

La declaración private es la declaración predeterminada de un miembro en C#. Significa que dicho miembro sólo es visible desde la propia clase y sus clases anidadas (que no derivadas), pero no desde variables de instancia (desde otros objetos que instancien una variable de esa clase). Constituye el pilar fundamental

del principio de la Encapsulación, y protege del acceso inadvertido a los miembros de una clase, permitiendo preservar la lógica interna definida en su creación.

Public

Aunque existe en todos los lenguajes orientados a objetos, su utilización se justifica raramente, puesto que atenta contra el primer principio de la OOP: la Encapsulación. Un miembro public es accesible sin restricciones, por lo que no hay ningún tipo de control sobre los valores que son asignados y/o devueltos.

Entre estos dos límites C# proporciona métodos mixtos de acceso, permitiendo distintos niveles:

Protected

Es idéntico a private, a excepción de que se permite que los miembros de las clases derivadas tengan acceso al mismo (al igual que las clases anidadas). Se presta bien para ofrecer funcionalidad a clases derivadas, sin permitir asignaciones inapropiadas, como en el caso de los miembros public. El siguiente código fuente ilustra perfectamente esta situación:

Ejemplo de declaraciones tipo Protected

```
class A
{
   protected int x = 123;
}

class B : A
{
   void F()
   {
      A a = new A();
      B b = new B();
      a.x = 10;   // Error
      b.x = 10;   // OK
   }
}
```

El fallo se produce debido a que la clase A no deriva de la clase B (sino al contrario), y el miembro x, está declarado en A, como ***protected***.

Internal

Es un modificador de acceso para tipos y miembros de tipos. Eso significa que el modificador es aplicable la declaración de clases, cuando queremos limitar su uso al proyecto en el que son declaradas. Los miembros internos sólo son accesibles dentro de archivos del mismo ensamblado.

El MSDN nos recuerda que "un uso habitual del acceso de tipo interno se da en el desarrollo basado en componentes, ya que permite a un grupo de componentes cooperar de manera privada sin estar expuesto al resto del código de la aplicación. Por ejemplo, una estructura para crear interfaces gráficas de usuario podría proporcionar clases ***Control*** y ***Form*** que cooperan mediante miembros con acceso de tipo ***internal***. Como estos miembros son internos, no están expuestos al código que utiliza la estructura".

Protected internal

Es el único modificador que admite más de una palabra reservada. El acceso se limita al proyecto actual o a los tipos derivados de la clase contenedora.

Static

Aunque ya nos hemos referido a él en artículos anteriores, recordemos que sirve para declarar un miembro que pertenece al propio tipo en vez de a un objeto específico. El modificador ***static*** se puede utilizar con campos, métodos, propiedades, operadores y constructores, pero no con indizadores, destructores o tipos.

Readonly

El modificador ***readonly***, indica, obviamente, que un miembro es de sólo lectura. Se diferencian de las declaraciones ***const*** que veremos a continuación en que no pueden ser calculadas en tiempo de compilación, generalmente, debido a que se precisa una instancia de un objeto para hacerlo, y que deben de ser asignadas, o bien en su declaración, o en el método constructor de la clase

Const

Se definen como campos cuyo valor puede ser determinado por el compilador en tiempo de compilación y que no se pueden modificar en tiempo de ejecución (son, de forma implícita, ***readonly***). Son, pues, más restrictivas que las variables ***readonly***, pero pueden definirse también a partir de otras constantes (pero no variables aunque éstas hayan sido declaradas previamente):

Ejemplo de constantes

```
const int x = 1;
const int y = x + 1;
int z = 2;
const w = z + y + 1; //Error: z es una variable
```

Propiedades

Sin duda, una de las novedades del lenguaje C# en comparación con sus similares (C++, Java) es la presencia de las propiedades, también llamadas campos inteligentes, que no son sino métodos de acceso a los campos de una clase, que se ven como campos desde el cliente de esa clase. Se trata de un mecanismo más elegante de acceso al estado[1] de una clase, y una de las aportaciones que el lenguaje C# realiza a la mejora de las técnicas de encapsulación en los lenguajes OOP

Las propiedades siguen la notación típica de C# para las estructuras de control, disponiendo de 2 modificadores de acceso: Get para la lectura y Set para la asignación de valores al campo (escritura). El ejemplo siguiente declara una propiedad Nombre de tipo lectura/escritura y otra propiedad FechaDeAlta de sólo lectura, que son accedidas posteriormente. Un intento de escribir en la propiedad fechaAlta, de sólo lectura, genera un error en tiempo de compilación:

Ejemplo de clase con propiedades

```
using System;
class ClaseConPropiedades {
        string nombre;
       System.DateTime fechaAlta;
       public string Nombre
             {
                    get { return nombre; }
                    set {nombre = value;       }
             }
       public DateTime FechaAlta
             {
                    get { return fechaAlta; }
             }
       public ClaseConPropiedades(string nombre) //El constructor
inicializa
                                                  // fechaAlta
       {
             fechaAlta = System.DateTime.Now;
              this.nombre = nombre;
       }
       static void Main()
       {
                    ClaseConPropiedades ccp = new
ClaseConPropiedades("Posadas");
                    Console.WriteLine("El cliente {0} se dio de alta
en {1}",
                                        ccp.Nombre, ccp.FechaAlta);
                    ccp.FechaAlta = new DateTime(2000, 1, 1, 0, 0, 0);
//Error
       }
}
```

Métodos

Los métodos de una clase constituyen su componente funcional, siendo aquello que las dota de un comportamiento. Siguiendo las definiciones formales que aporta el MSDN diremos que un método es "un miembro que implementa un cálculo o una acción que pueden ser realizados por un objeto o una clase. Los métodos tienen una lista de parámetros formales (que puede estar vacía), un valor devuelto (salvo que el tipo de valor devuelto del método sea void), y son estáticos o no estáticos. El acceso a los métodos estáticos se obtiene a través de la clase. El acceso a los métodos no estáticos, también denominados métodos de instancias, se obtiene a través de instancias de la clase".

Métodos constructores

Recordemos que un método constructor es aquel que se llama siempre que se crea una instancia de una clase. Debe de tener el mismo nombre de la clase, pero admite diferentes signaturas, dependiendo de los distintos conjuntos de parámetros que queramos pasar al constructor. En C# existen 3 tipos de constructores: **de instancia**, **private** y **static**.

Un constructor de instancia se utiliza para crear e inicializar instancias, y es –generalmente- el constructor predeterminado. La sintaxis completa de un constructor de este tipo es:

[atributos] [modificadores] identificador ([lista-de-parámetros-formales]) [inicializador]

{ cuerpo-del-constructor }

donde los tres parámetros más significativos son:

- Los modificadores permitidos, que pueden ser **extern** ó cualquiera de los 4 modificadores de acceso
- identificador, que es el nombre de la clase, e
- inicializador que es una llamada opcional a la clase base (mediante :***base***(<lista>)) o una referencia a la propia clase, expresada mediante ***this***(<lista>), en caso de que deseemos llamar a otro constructor de la misma clase, antes de ejecutar el cuerpo del constructor que hace la llamada.

Dando por sentado que la sintaxis de instanciación de objetos en C# es:

<clase> <objeto> = new <clase> (argumentos del constructor),

debemos tener en cuenta que en un método constructor se cumplen siempre los siguientes supuestos:

- Un método constructor siempre tiene el mismo nombre de la clase a la que pertenece.
- Un método constructor no devuelve ningún tipo de dato
- Un método constructor admite los modificadores: **public, protected, internal, private, extern**
- Un método constructor admite atributos.
- Se dice que la lista de parámetros formales opcional de un constructor de instancia está sujeta a las mismas reglas que la lista de parámetros formales de un método. Dicha lista de parámetros constituye la firma del método y se encarga de discriminar la llamada correspondiente, cuando el constructor está sobrecargado.
- Todo método constructor (excepto ***System.Object***) efectúa una llamada implícita previa a la clase base del constructor, inmediatamente antes de la ejecución de la primera línea del constructor. Se conoce estas llamadas como inicializadores de constructor (***Constructor Initializers***).
    - Estos inicializadores de constructor, si son explícitos (si se indican como tales en el código fuente), pueden presentarse en dos formas, dependiendo de la palabra reservada que usemos en la llamada: base o this.
        - Si utilizamos ***base*** estaremos llamando al constructor de la clase base de la que hereda nuestra propia clase.
        - Si usamos ***this***, podemos efectuar una llamada interna desde un constructor a otro constructor dentro de la clase.

Veamos un ejemplo de esta situación:

Ejemplo de llamadas a constructores

```
using System;
class A
{
    public A()
  {
        Console.WriteLine("A");
    }

    public A(int x)
    {
        Console.WriteLine("A = {0}", x);
    }
}
class B : A
{
    public B(int x)
  {
        Console.WriteLine("B = {0}", x);
    }
}
class Pruebas
{
    public static void Main()
    {
        B b = new B(3);
    }
}
```

Los constructores ***private*** son un tipo especial de constructor de instancias que no es accesible desde fuera de la clase. Suelen tener el cuerpo vacío, y no es posible crear una instancia de una clase con un constructor ***private***. Los constructores ***private*** son útiles para evitar la creación de una clase cuando no hay campos o métodos de instancia (son todos ***static*** como, por ejemplo, la clase ***Math***), o cuando se llama a un método para obtener una instancia de una clase.

Ejemplo de Constructor private

```
public class ConstructorPrivate
{
   private ConstructorPrivate() {} // Cuerpo vacío y miembros
estáticos
   public static int contador;
   public static int IncrementarContador ()
   {
      return ++contador;
   }
}

class Pruebas
{
   static void Main()
   {
      // La llamada siguiente a ConstructorPrivate(comentada) genería
un error
      // por que no es accesible
      // ConstructorPrivate cs = new ConstructorPrivate();   // Error
      ConstructorPrivate.contador = 100;
      ConstructorPrivate.IncrementarContador ();
      Console.WriteLine("Valor: {0}", ConstructorPrivate.contador);
   }
}
--------------------
Salida en ejecución:
--------------------
Valor: 101
```

Los constructores ***static*** son un tipo especial de constructor que se utiliza para inicializar una clase (no para crear una instancia de ella). El MSDN nos recuerda que "*se llama automáticamente para inicializar la clase antes de crear la primera instancia o de hacer referencia a cualquier miembro estático*". Se declara de la siguiente forma:

[atributos] static identificador ( ) { cuerpo-del-constructor }

Tienen las siguientes características especiales:

Un constructor ***static*** no permite modificadores de acceso ni tiene parámetros.

- Es llamado automáticamente para inicializar la clase antes de crear la primera instancia o de hacer referencia a cualquier miembro estático.
- El constructor ***static*** no puede ser llamado directamente.
- El usuario no puede controlar cuando se ejecuta el constructor ***static*** en el programa.
- Los constructores ***static*** se utilizan normalmente cuando la clase hace uso de un archivo de registro y el constructor escribe entradas en dicho archivo"

Un ejemplo de su uso sería el siguiente:

Ejemplo de Constructor static

```
using System;
class A
{
   // Constructor estático
   static A ()
   {
      Console.WriteLine("Llamado el constructor estático");
   }

   public static void F()
   {
      Console.WriteLine("Llamado F().");
   }
}

class MainClass
{
   static void Main()
   {
      A.F();
   }
}
```

Con ello se consigue forzar la ejecución del constructor, sea cual sea el miembro de la clase que se llame.

Métodos destructores y el Garbage Collector

En C#, y, a diferencia de la mayoría de los lenguajes que soportan algún tipo de orientación a objetos, los métodos destructores no debieran ser codificados explícitamente, salvo cuando el tipo encapsule recursos no administrados como ventanas, archivos y conexiones de red. Esto es debido a que la "recolección de basura" (la destrucción de un objeto y posterior reorganización de la memoria ocupada por él) es realizada automáticamente por el ***Garbage Collector***, que es parte vital del entorno de ejecución de .NET.

Eso significa, que, aunque exista un método para forzar la llamada al destructor de un tipo (que recibe el nombre de **Finalize**()) su llamada expresa sólo tiene sentido cuando los recursos externos utilizados por el tipo son de peso, o tienen alguna característica especial que les hace susceptibles de una destrucción explícita. En su lugar, los lenguajes del .NET Framework prefieren invocar a un método predeterminado **Dispose**(), que indica al GC que el objeto en cuestión puede ser destruido y el espacio en memoria ocupado por él, reutilizado, en cuanto su programación predeterminada lo aconseje. Otra razón en contra de la utilización explícita de un método **Finalize**() es su penalización en el rendimiento.

La sintaxis de un destructor es similar a la de su constructor opuesto:

[atributos] ~ identificador ( ) { cuerpo-del-destructor }

El editor de Visual Studio .NET se encarga de situar los métodos adecuados de finalización, por lo que no vamos a hacer más hincapié en este punto.

Modificadores de métodos

Son palabras reservadas que califican el comportamiento de un método tanto desde un punto de vista funcional como de arquitectura. Los principales modificadores de métodos aparecen explicados brevemente en la siguiente tabla.

Modificador	Descripción
Static	El método pertenece al propio tipo (clase) en lugar de a un objeto específico. No pueden llamarse desde una variable de instancia.
Abstract	El método es un miembro de una clase abstracta. (Si la clase no se declarara como abstracta, esta opción no está disponible). No está permitido heredar de un tipo abstracto. Su función se limita a la definición de jerarquías de clases.
Virtual	La implementación del método puede cambiarse mediante un miembro de reemplazo (**override**) de una clase derivada. Pueden llamarse igual que los métodos estándar.
Extern	Sólo se establece su declaración formal. La implementación es externa. Se usa principalmente para declarar funciones de una librería externa a .**NET Framework**.
Override	Proporciona una nueva implementación de un miembro virtual heredado de una clase base.
New	Oculta explícitamente un miembro heredado de una clase base. Se diferencia de **override** en que la ocultación impide la llamada al mismo método en las clases antecesoras.
	Modificadores Especiales
Abstract	Aplicable a una clase y también a un método cuya clase haya sido declarada de igual forma, permite definir jerarquías de clases, sin proporcionar implementaciones de los métodos.
Sealed	Aplicable exclusivamente a las clases, establece que la clase no se puede heredar, solo se permite la instanciación de objetos de la clase.

Hay que tener en cuenta que algunos modificadores se pueden combinar, pero otros son mutuamente excluyentes. Por ejemplo, se pueden establecer combinaciones como **extern static**, **new virtual**, y **override abstract**.

Sobrecarga de Operadores (Operator Overloading)

Hemos visto el funcionamiento de la sobrecarga aplicada a los métodos, pero ésta no es la única forma de sobrecarga que permite el lenguaje C#: también podemos establecer cómo deseamos que sea el comportamiento de nuestros propios tipos cuando les aplicamos los operadores ariméticos clásicos, como +, -, *, etc, y –en genera—cualquier operador.

Como se ha hablado mucho en pro y en contra del uso de la sobrecarga de operadores en los lenguajes de programación, me gustaria citar una opinión a la que me adhiero, y que puede leerse en [Gunnerson 2000]: "Sólo se debería sobrecargar un operador si así se obtiene un código más claro. Y observe que el texto dice más claro, no más corto". O como recomienda la propia documentación del MSDN: "La norma para saber cuándo sobrecargar es muy simple. Si el usuario esperaría poder realizar esa operación, se debería sobrecargar

Operadores sobrecargables en C#	
Operadores Unarios	+, -, !, ~, ++, --, true, false
Operadores Binarios	+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=

Hay que observar que ciertos operadores no son sobrecargables, como el operador de asignación (=), si bien se produce una sobrecarga implícita al sobrecargar los operadores binarios aritméticos básicos (+, -, *, /) y utilizarlos posteriormente en expresiones compuestas del tipo x += 1.

Por ejemplo, imaginemos que se dispone de una clase que maneja números complejos de la forma x+yi, donde i = . Como el lector sabrá, tales números admiten una representación en el plano real que especifica, en coordenadas cartesianas los valores de las componentes real e imaginaria del número complejo (respectivamente, x e y). Dado que la operación de suma de complejos está definida en el plano real, sería interesante poder manejar dicha suma mediante la sobrecarga del operador + en nuestra clase.

Podríamos escribir el siguiente código:

Ejemplo de Sobrecarga

```
// Ejemplo sobrecarga con números complejos
using System;

public struct Complejo
{
public int real;
public int imaginaria;

public Complejo(int real, int imaginaria)
{
this.real = real;
this.imaginary = imaginaria;
}

// Declarar el operador de sobrecarga (+), los tipos
// que se pueden agregar (dos objetos Complejo), y el
// tipo de valor devuelto (Complejo):
public static Complejo operator +(Complejo c1, Complejo c2)
{
return new Complejo(c1.real + c2.real, c1.imaginaria + c2.imaginaria);
}
// Reemplazar el método ToString para mostrar adecuadamente número
complejo
public override string ToString()
{
return(String.Format("{0} + {1}i", real, imaginaria));
}

public static void Main()
{
Complejo num1 = new Complejo(2,3);
Complejo num2 = new Complejo(3,4);

// Agregar dos objetos Complejo (num1 y num2) mediante el operador más
sobrecargado
Complejo sum = num1 + num2;

// Imprimir los números y la suma usando el método reemplazado
ToString:
Console.WriteLine("Primer número complejo: {0}",num1);
Console.WriteLine("Segundo número complejo: {0}",num2);
Console.WriteLine("Suma de los dos números: {0}",sum);

}
}
```

No obstante, conviene recordar lo dicho al principio, la sobrecarga debe de sopesarse adecuadamente, para evitar lo que algunos llaman "una solución en busca de un problema", o sea, utilizar una característica a toda costa, tenga o no sentido en la solución que intentamos construir.