20 consejos y conceptos imprescindibles para programadores de

# C# y .NET

campus MVP

www.campusMVP.es

# Índice

# Introducción

Te damos la bienvenida. Este libro está **dirigido a programadores que ya saben algo sobre la plataforma .NET**, aunque estés empezando. Al menos debes conocer las estructuras básicas, los tipos de datos existentes y cuestiones de base similares.

En algunos de estos 20 capítulos se explican algunas cuestiones de "bajo nivel" y por lo tanto un poco "áridas". Muchas es difícil encontrarlas explicadas con detalle en libros de texto o en cursos, por lo que les sacarás mucho partido. Sirven para cubrir ciertos **conocimientos importantes que todo programador profesional debiera poseer**, pero que muchos desconocen. Te servirán para **comprender mejor el funcionamiento interno** de la plataforma .NET y **marcarán diferencias entre tú y otros programadores** que no tienen los conceptos tan claros.

Algunas cuestiones sobre el funcionamiento del lenguaje o de la plataforma son bastante sutiles. Lo mejor es **que hagas pruebas en tu propio equipo a medida que lees** algunos de los capítulos, porque así podrás comprobar de primera mano lo que se indica en el texto. Sería muy recomendable.

Verás también que el texto contiene multitud de enlaces a recursos adicionales para aprender más. Púlsalos cuando los veas. No hay ninguno de relleno y todos ellos se han puesto porque son interesantes. Aumentarás tus horizontes también con ellos.

Esperamos que este libro te sirva para profundizar un poco más en algunas cuestiones de base que, si bien en muchos casos no son del día a día, te servirán para sobresalir en tu profesión.

¡Comenzamos!

# Pregunta-Trampa: El lenguaje C# y la plataforma .NET no son lo mismo

Por **Alberto Población**

Hace poco un alumno me hizo una consulta acerca de una pregunta que le habían planteado en un ejercicio académico. Considero interesante reproducirla, porque nos permite clarificar y distinguir lo que es el propio lenguaje C# y las librerías que habitualmente utilizamos desde C#.

## Enunciado

> Indicar cómo se puede escribir un texto dentro de un archivo en disco utilizando únicamente el lenguaje C#. Se darán puntos extra a quien responda en una sola línea.

## Respuesta breve

**Es imposible.**

## Respuesta larga

Se trata de una pregunta-trampa. El lenguaje **C# no define instrucciones de entrada/salida.** Escribir un archivo en disco es una operación de salida. En consecuencia, no se puede hacer utilizando *únicamente* C#.

Para escribir el fichero, **será necesario recurrir a alguna biblioteca externa** que sea capaz de realizar las operaciones de entrada/salida. Bajo circunstancias normales, típicamente se utilizarán las bibliotecas de la *Framework Class Library* (FCL) incluidas en el Framework de .Net. Para escribir en un archivo en disco, usaremos las clases contenidas en el espacio de nombres System.IO dentro de la biblioteca System.DLL.

Para ello, será necesario añadir una referencia a esa biblioteca. Si el programa se genera con Visual Studio, de manera predeterminada se añade automáticamente la referencia a System.DLL. Lo mismo sucede si compilamos desde línea de comandos con CSC.EXE.

Una vez referenciada la biblioteca, es muy sencillo escribir el texto en el archivo:

```
System.IO.File.WriteAllText(@"c:\ruta\nombre.txt",
"Contenido");
```

Nótese que hemos resuelto en una sola línea el problema inicialmente planteado... pero no lo hemos hecho utilizando *únicamente* C#. Aunque nosotros sólo hemos *tecleado* código en C#, nos estamos apoyando en una biblioteca externa, que por dentro utiliza (directa o indirectamente) un bloque de código que no está escrito en C#.

## Enseñanza

Con frecuencia alguien plantea en los foros públicos una consulta sobre programación insistiendo en que tiene que hacerlo solo con C#. En general, lo que en realidad quieren decir es que desean hacerlo **solo con las librerías del Framework,** sin adquirir ninguna biblioteca de terceras partes. Pero recordemos que esas librerías no forman parte del lenguaje C#, simplemente suelen usarse conjuntamente con C#, pero son independientes del lenguaje.

> Una cosa es el lenguaje (C#, Visual Basic o cualquiera de los muchos disponibles en .NET) y otra muy diferente es la plataforma .NET.
> Es necesario tenerlo muy claro.

# C# o VB.NET ¿qué lenguaje debo aprender?

Por **José Manuel Alarcón**

Cuando alguien se quiere iniciar en la programación y se decide por la plataforma .NET, enseguida le surge la duda: ¿qué lenguaje debo escoger? Y es que la plataforma .NET permite programar en decenas de lenguajes diferentes, si bien los dos principales, de entre los que proporciona Microsoft son Visual Basic .NET y C# (se pronuncia "C Sharp", por cierto).

Vamos a analizar diversos factores de peso, detallando un ganador en cada uno, que espero que te ayuden a tomar una decisión fundamentada. Aunque yo, personalmente, hace unos años que lo tengo claro...

¡Allá vamos!

## 1.- Capacidades de los lenguajes

Evidentemente si vamos a invertir muchas horas y esfuerzo personal en aprender un lenguaje, no querremos apostar por uno que tenga características inferiores. Nos interesará aprender aquél que tenga mayor potencia y permita hacer más cosas.

En ese sentido la buena noticia es que **ambos lenguajes tienen idénticas capacidades**.

En **primer lugar**, porque lo que hacemos con ellos es programar para la plataforma .NET **y la mayor parte de la funcionalidad se proporciona a través del framework.** Los lenguajes no son más que una forma de "pegar" entre sí la funcionalidad de base que extraemos de .NET, ya que **todas las bibliotecas y capacidades de la plataforma están al alcance de cualquier lenguaje** válido para ésta.

Al final, **ambos se compilan a** Lenguaje Intermedio (CIL) de .NET, e incluso es posible traducir el código de uno a otro y viceversa. De hecho **uno de los objetivos de Microsoft es que sean totalmente equivalentes** y que lo que se pueda hacer con uno, se pueda hacer también con el otro. Es más, la última versión de los compiladores para la plataforma .NET (llamado Roslyn) tiene el compilador de VB escrito en VB y el de C# escrito en C#, de modo que ambos sean funcionalmente equivalentes.

Aun así existen **diferencias en la sintaxis de cada lenguaje** que hacen que sea más o menos difícil trabajar con ellos. En general el código **Visual Basic es más fácil de leer**

al no usar llaves (tan típicas de otros lenguajes), aunque esto seguro que más de uno me lo discute. Además, ofrece "azúcar sintáctico" para ayudarnos con la escritura de código como, por ejemplo, entre otras muchas cosas:

- **El espacio de nombres My**: proporciona acceso rápido y directo a multitud de características comunes que de otro modo tendrías que localizar a través de sus espacios de nombres y referencias. Ayuda a acelerar el desarrollo, aunque todo lo que ofrece My se puede conseguir en C# también, pero no es tan directo.
- **WithEvents y la cláusula Handles**: que facilitan enormemente el uso de eventos.
- **With..End With:** estructura con la que podemos acceder a las propiedades y métodos de un objeto que vayamos a usar con frecuencia sin tener que estar escribiendo su nombre todo el rato, acelerando la escritura de código.
- **Importación de espacios de nombre a nivel de proyecto**: en C# solo se pueden importar por archivo, así que de este modo todos los espacios de nombres comunes se gestionan en un único sitio.
- **La cláusula When en la gestión de excepciones**: que es algo más que azúcar sintáctico y en este caso gana VB.NET.
- **Literales XML**: para facilitar la escritura de código XML en tus programas

Y estos son solo unos pocos ejemplos. Hay más.

Por otro lado hay algunas cosas de C# que no están disponibles en Visual Basic, como por ejemplo:

- **Bloques unsafe:** que nos permiten trabajar a más bajo nivel con punteros, de manera parecida a como se hace en C++, y saltarnos la seguridad de tipos.
- **Palabras clave checked y uncheked**: para saltarse la seguridad de tipos en desbordamiento.
- **Interfaces parciales:** C# permite definir interfaces en diversos archivos, al igual que clases, estructuras y métodos. VB solo estos tres últimos.
- **Azúcar sintáctico:** y sí, C# también tiene sus propias facilidades para escribir código que no tiene VB, como por ejemplo: cadenas multi-línea y escapeadas, comentarios multi-línea, implementación implícita de interfaces o el operador ?? para comprobar nulos (ver más adelante en este documento).

En este enlace tienes una buena referencia que **compara una a una las características de ambos lenguajes** (y alguno más como F# y C++).

Podríamos seguir hablando de este tema largo y tendido, pero como resumen podemos decir que, en general, **escribir código con VB.NET es más fácil y además más permisivo (menos estricto) que escribir código C#**. Y todo eso sin perder potencia ya que **son funcionalmente equivalentes**. En ese sentido es más fácil iniciarse a la programación en Visual Basic que en C#.

Aun así, una vez que coges experiencia en cualquiera de los dos, **elegir uno u otro es más una cuestión "estética"** y de preferencias personales que una diferencia real.

**Ganador**: Visual Basic, por facilidad de uso.

## 2.- Aspectos "sociales"

Uno de los aspectos más importantes a tener en cuenta a la hora de decidirse por un lenguaje de programación, no es solamente su capacidad o facilidad de uso, sino lo que yo llamo sus "aspectos sociales". Con esto me refiero a **la adopción que hay por parte de la comunidad de uno u otro lenguaje**. Esto es importante porque **mide el grado de ayuda** que nos vamos a encontrar para cada lenguaje, así como **la base de código Open Source con la que podemos contar** para acometer algunos proyectos.

> Los "aspectos sociales" de un lenguaje miden, en definitiva, lo solos o acompañados que estamos cuando trabajamos y nos encontramos con un problema o una dificultad.

Por un lado tenemos **el uso que se le da a cada lenguaje por parte de los profesionales.** No es lo mismo llegar a una empresa programando en VB y encontrarte que todos tus compañeros programan en C#, que lo contrario. Eso es difícil de estimar y va a depender de dónde caigas, claro, pero si tuviésemos unas estadísticas generales podría ser un dato importante. Sobre esto no hay datos fiables. Sin embargo, una cosa a tener en cuenta es que en la mayor parte de los institutos y las facultades de informática se enseña o bien C++ o bien Java. Debido a ello **muchas personas se sienten más cómodas usando la sintaxis de C#**, casi idéntica en todo a la de estos, y luego **en el trabajo tienden a utilizar C# y no Visual Basic**. Pero como digo, es difícil saberlo.

Una buena pista nos la pueden dar **los índices de la industria**, que también son inexactos pero proporcionan información interesante que marca tendencias. Uno de los índices más populares, que se actualiza mensualmente, es el de **TIOBE de popularidad de lenguajes de programación**, que se basa en analizar resultados de búsquedas en más de 110 buscadores en todo el mundo, tratando de determinar qué lenguajes son más populares entre los desarrolladores en función de lo que se publica sobre ellos. En Febrero de 2016 C# ocupaba el 4º puesto con un 4,4% de popularidad, mientras que Visual Basic .NET estaba en el 7º puesto con un 2,454%. Es decir, aparentemente **C# es el doble de popular que VB.NET** hoy en día. Hace 5 años la cosa no era así e iban prácticamente parejos, pero está claro que en los últimos tiempos la ascensión de C# (o el declive de VB) en cuanto a preferencias de los programadores es imparable. Es más, de esto no tengo números, pero es evidente que hay muchos más blogs y artículos para C# que para VB.

> La gracia, por cierto, es que Visual Basic (es decir, la versión original de lenguaje) cuya última versión es de hace más de 15 años, está justo detrás en el decimosegundo puesto con un 1,885% de popularidad ¡e incluso ha subido respecto al año anterior! Que cada uno saque sus propias conclusiones respecto a este hecho.

Según otros índices los resultados son incluso peores para Visual Basic. Por ejemplo, **el índice PYPL** que se basa en la disponibilidad de cursos y tutoriales disponibles para cada lenguaje (un indicador de demanda), C# es el 4º lenguaje más popular con un 8,9%, y VB.NET está en decimotercer lugar con un 1.8% y cayendo bastante.

| Feb 2016 | Feb 2015 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 2 | ▲ | Java | 21.145% | +5.80% |
| 2 | 1 | ▼ | C | 15.594% | -0.89% |
| 3 | 3 | | C++ | 6.907% | +0.29% |
| 4 | 5 | ▲ | C# | 4.400% | -1.34% |
| 5 | 8 | ▲ | Python | 4.180% | +1.30% |
| 6 | 7 | ▲ | PHP | 2.770% | -0.40% |
| 7 | 9 | ▲ | Visual Basic .NET | 2.454% | +0.43% |
| 8 | 12 | ⇈ | Perl | 2.251% | +0.86% |
| 9 | 6 | ▼ | JavaScript | 2.201% | -1.31% |
| 10 | 11 | ▲ | Delphi/Object Pascal | 2.163% | +0.59% |
| 11 | 20 | ⇈ | Ruby | 2.053% | +1.18% |
| 12 | 10 | ▼ | Visual Basic | 1.855% | +0.14% |

*Índice TIOBE de Febrero de 2016*

**Worldwide**, Feb 2016 compared to a year ago:

| Rank | Change | Language | Share | Trend |
|---|---|---|---|---|
| 1 | | Java | 24.2 % | +0.3 % |
| 2 | ↑ | Python | 11.9 % | +1.2 % |
| 3 | ↓ | PHP | 10.7 % | -0.8 % |
| 4 | | C# | 8.9 % | +0.1 % |
| 5 | | C++ | 7.6 % | -0.5 % |
| 6 | | C | 7.5 % | +0.1 % |
| 7 | | Javascript | 7.3 % | +0.3 % |
| 8 | | Objective-C | 5.0 % | -0.9 % |
| 9 | ↑↑ | Swift | 3.0 % | +0.4 % |
| 10 | | R | 2.9 % | +0.3 % |
| 11 | ↓↓ | Matlab | 2.8 % | -0.3 % |
| 12 | | Ruby | 2.3 % | -0.2 % |
| 13 | | Visual Basic | 1.8 % | -0.4 % |
| 14 | | VBA | 1.5 % | +0.1 % |

*Índice PYPL de Febrero de 2016*

Si consideramos la popularidad dentro de los proyectos Open Source, entonces GitHub es un buen sitio donde mirar, ya que actualmente es donde se "cuece" todo lo que

tiene que ver con este movimiento de software libre. Según los datos de GitHub el lenguaje más popular con mucha diferencia en 2015 fue JavaScript, pero de los dos que nos ocupan, **el único que aparece siquiera en los rankings es C#** (en 8º lugar).

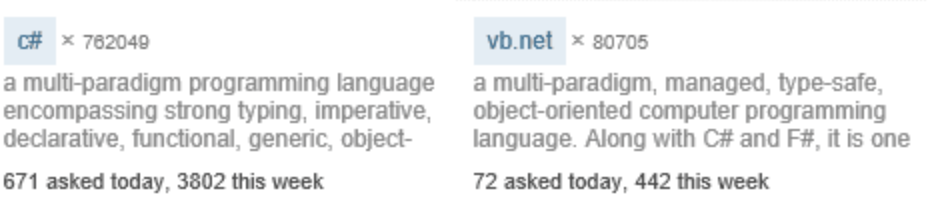
1. JavaScript
2. Java
3. Ruby
4. PHP
5. Python
6. CSS
7. C++
8. C#
9. C
10. HTML

*Lenguajes de programación ordenados por popularidad en Github en 2015*

Si consideramos un repositorio de código fuente con más sesgo hacia tecnologías Microsoft, entonces debemos referirnos a Codeplex. En este repositorio de código fuente de tecnología Microsoft, si buscamos a día de hoy los proyectos Open Source más descargados (que son los que realmente tienen algo detrás y no están en planificación o vacíos), vemos que C# tiene 3.154 proyectos mientras que Visual Basic tan solo 319. O sea, gana C# por goleada:


*Lenguajes de programación ordenados por popularidad en Codeplex en 2015*

Finalmente, y para tratar de ser exhaustivos, vamos a ver qué pasa con StackOverflow. Este es el sitio más popular del mundo para resolver dudas sobre cualquier cuestión relacionada con la programación. Es tan popular que hay quien dice, con muy mala uva, que si mañana lo cerraran media fuerza laboral del sector TIC se iría al paro pues no sabrían qué hacer ;-) Bien, si vemos en este sitio el número de preguntas que se hacen en cada lenguaje, tenemos los siguientes resultados:




*Popularidad en StackOverflow*

Es decir, la diferencia es de casi 10 a 1 a favor de C#.

**Ganador**: C# por goleada. Con C# tendremos mucha más probabilidad de ayuda, código fuente de base, aprendizaje, etc...

# 3.- Ámbitos que abarcan

Esta es otra gran pregunta que debemos hacernos. No solo llega con ver cuál tiene más capacidad o con cuál tenemos más ayuda. Tenemos que ver también con cuál podremos hacer más cosas y seremos capaces de abarcar más ámbitos.

En la actualidad la plataforma .NET está presente en el escritorio, en los servidores, en la nube, en dispositivos móviles, etc... Pero no significa que podamos usar cualquier lenguaje. En este caso solo C# nos deja abarcar cualquier ámbito en el que esté presente .NET, y en concreto **el mundo móvil está vedado para Visual Basic .NET.**

Sí, podemos programar para Windows Phone con C# y con VB.NET, pero **si queremos abarcar cualquier plataforma móvil (iOs, Android...)** o tablet usando .NET, es decir, usando **Xamarin**, entonces el lenguaje que **debemos usar obligatoriamente es C#.** Y éste es un punto importantísimo que no debemos olvidar. Aunque ahora no programes para móviles ten claro que si te dedicas profesionalmente a esto, más temprano que tarde lo vas a tener que hacer. Y si para ello puedes evitar aprender un nuevo lenguaje, mejor ¿no?

Debemos tener en cuenta también que, desde hace unos años, prácticamente **todas las nuevas tecnologías que va lanzando Microsoft se crean con C#.** A esto debemos añadir que además ahora son Open Source, lo cual es un argumento de peso, y si no, echa un vistazo a su código fuente:

- .NET Core
- ASP.NET MVC
- Entity Framework
- Json.NET

Y así podríamos seguir todo el día... Puedes encontrar aquí una lista exhaustiva.

**Ganador**: C#. Podrás hacer muchas más cosas a largo plazo.

# 4.- Otros factores

Finalmente hay que considerar otros factores, quizá menos importantes, pero que debemos tener en cuenta igualmente a la hora de decidirnos.

Por ejemplo, **de qué base partes** para aprender.

**Si vienes de programar con Visual Basic 6 clásico**, en los '90 (hay MUCHÍSIMA gente que viene de ahí y se está pasando aún ahora), seguro que la sintaxis de VB.NET te va a resultar más sencilla y rápida de asimilar. De hecho, el lenguaje se diseñó para que fuera compatible lo máximo posible con VB clásico. Sin embargo, hay bastantes diferencias. Aunque la mayoría son cosas tontas (por ejemplo en VB.NET debes poner paréntesis para llamar a todas las funciones y en VB clásico no) pueden suponerte un problema más que una ventaja. Si tienes que mantener código en las dos versiones del lenguaje (clásica y moderna) te encontrarás que cometes muchos más errores por estas pequeñas cosas, lo cual es desesperante. Al final casi te resultará más sencillo marcar mucho más la diferencia entre los dos ambientes usando C# para .NET. Al principio te resultará más complicado, pero cuando te acostumbres cambiarás fácilmente de contexto, irás mucho más rápido y tendrás mayor productividad.

Y no lo digo por decir: en su día, hace muchos años, yo mismo pasé por esta situación, así que lo digo por experiencia propia ;-)

Si vienes de programar con C, C++, Java o similar, la sintaxis de C# te resultará mucho más natural y cómoda. Aunque claro, también te puede pasar algo parecido a lo de VB clásico con VB.NET, aunque aquí las diferencias de sintaxis son mucho mejores. De hecho, hay fragmentos sencillos de código que si te los ponen delante no sabrías decir si son C# o Java, de tan parecidos que son.

Otra cuestión importantísima **es la percepción del lenguaje en el mercado laboral.** Se dice que no hay que juzgar un libro por las tapas y que no debes dejarte llevar por la corriente o las modas. Pero al final lo hace todo el mundo. Y si no, ¿por qué cualquier cosa que lance Apple al mercado se vende como si no hubiese un mañana, independientemente de su calidad o bondad técnica? Pues por eso, por el factor "cool" de la marca.

En programación pasa exactamente lo mismo. Hay mucha "pijería". Aunque nos cueste reconocerlo, es la realidad. Y una cosa obvia es que C# hoy por hoy tiene un factor "cool" que no posee Visual Basic .NET. C# se relaciona con lo moderno y VB.NET con un lenguaje de los años 90 que se ha adaptado para la actualidad. En tu currículo va a

lucir más C# que Visual Basic. Da igual que VB.NET sea igualmente capaz que C# y que de hecho sea más fácil de leer y de escribir. Puede que no tenga sentido técnico alguno, pero es un hecho.

Obviamente no lo puedo demostrar con datos fehacientes ya que es una cuestión cualitativa, no cuantitativa. Pero si no me crees puedes hacer la prueba y preguntar por ahí o mandar CV iguales con VB y con C# a ver qué pasa ;-)

**Ganador:** C#

# Conclusiones

Seguro que más de un fan de Visual Basic ahora mismo está contando hasta 10 antes de escribir un comentario en total descuerdo con muchas cosas de las que digo. ¡Hey!, yo mismo fui programador de Visual Basic muchos años y me ha dado muchas alegrías. Pero los hechos son los que son:

- Ambos lenguajes son igual de capaces, y de hecho VB.NET es más fácil de leer y puede ser más productivo.
- Pero C# gana con mucha diferencia a Visual Basic .NET en los aspectos sociales del lenguaje (más código y ayuda disponibles)
- C# abarca más ámbitos que VB.NET, sobre todo en lo que respecta a programación multi-plataforma para móviles y tabletas.
- Es más fácil confundirse escribiendo código Visual Basic que C# si vienes de otro lenguaje.
- El factor *cool* está a favor de C#, por lo que te resultará más fácil encontrar trabajo o que te consideren en las empresas (y en ciertos círculos técnicos).

Creo que todos los puntos anteriores que desarrollo en el artículo son bastante objetivos, o al menos así lo he procurado, y creo que **hoy por hoy aprender C# es una mucho mejor apuesta de futuro que aprender Visual Basic .NET.**

# Tipos por valor y por referencia

por **José Manuel Alarcón**

Antes de empezar, necesitamos comprender dos conceptos importantes que paso a resumir de manera sencilla:

- **La pila o "stack":** es una zona de memoria reservada para almacenar información de uso inmediato por parte del hilo de ejecución actual del programa. Por ejemplo, cuando se llama a una función se reserva un bloque en la parte superior de esta zona de memoria (de la pila) para almacenar los parámetros y demás variables de ámbito local. Cuando se llama a la siguiente función este espacio se "libera" (en el sentido de que ya no queda reservado) y puede ser utilizado por la nueva función. Es por esto que si hacemos demasiadas llamadas anidadas a funciones (en recursión, por ejemplo) podemos llegar a quedarnos sin espacio en la pila, obteniendo un "stack overflow".
- **El montón o "heap":** es una zona de memoria reservada para poder asignarla de manera dinámica. Al contrario que en la pila no existen "normas" para poder asignar o desasignar información en el montón, pudiendo almacenar y eliminar datos en cualquier momento, lo cual hace más complicada la gestión de la memoria en esta ubicación.

Los tipos de datos llamados **"por valor"** son tipos sencillos que almacenan un dato concreto y que **se almacenan en la pila**. Por ejemplo, los tipos primitivos de .NET como *int* o *bool*, las estructuras o las enumeraciones. Se almacenan en la pila y se copian por completo cuando se asignan a una función. Por eso cuando se pasa un tipo primitivo a una función, aunque lo **cambiemos** dentro de ésta, el cambio no se ve reflejado fuera de la misma.

Los **tipos "por referencia"** son todos los demás, y en concreto todas las clases de objetos en .NET, así como algunos tipos primitivos que no tienen un tamaño determinado (como las cadenas). Estos tipos de datos se alojan siempre en el montón, por lo que la gestión de la memoria que ocupan es más compleja, y el uso de los datos es menos eficiente (y de menor rendimiento) que con los tipos por valor.

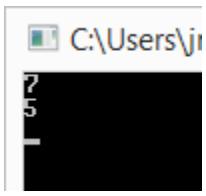
Bien. Ahora que ya conocemos de manera superficial estos conceptos, veamos qué pasa con los tipos por valor y por referencia al usarlos en un programa escrito con C#.

Si pasamos un tipo por valor a una función y lo modificamos dentro de ésta, el valor original permanecerá inalterado al terminar la llamada.

Por ejemplo:

```
public int Suma2(int n) {
  n = n+2;
  return n;
}

int i = 5;
Console.WriteLine(Suma2(i));
Console.WriteLine(i);
Console.ReadLine();
```

En este caso definimos una función que simplemente le suma 2 al parámetro que se le pase (de una forma poco eficiente, eso sí) transformando el valor que se le pasa. Se podría pensar que ya que la función cambia el valor que le hemos pasado, cuando mostremos por pantalla posteriormente el valor de **i**, éste debería ser 7. Sin embargo vemos que, aunque se ha cambiado dentro de la función, sigue siendo 5:



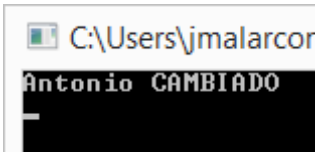

Esto es debido a que los números enteros son tipos por valor y por lo tanto se pasa una copia de los mismos a la pila de la función que se llama, no viéndose afectado el valor original.

Sin embargo, si lo que le pasamos a la función es un objeto (por lo tanto, un tipo por referencia), veremos que sí podemos modificar los datos originales.

Por ejemplo, si ahora definimos este código:

```
public class Persona
{
  public string Nombre;
  public string Apellidos;
  public int Edad;
}

public static void CambiaNombre(Persona per) {
  per.Nombre = per.Nombre + " CAMBIADO";
}

Persona p = new Persona();
p.Nombre = "Antonio";
p.Apellidos = "López";
p.Edad = 27;

CambiaNombre(p);
Console.WriteLine(p.Nombre);
Console.ReadLine();
```

Lo que hacemos en el fragmento anterior es definir una clase persona muy sencilla y pasársela a una función que modifica su nombre. Cuando ejecutemos el código se verá por pantalla el nombre de la persona cambiado:



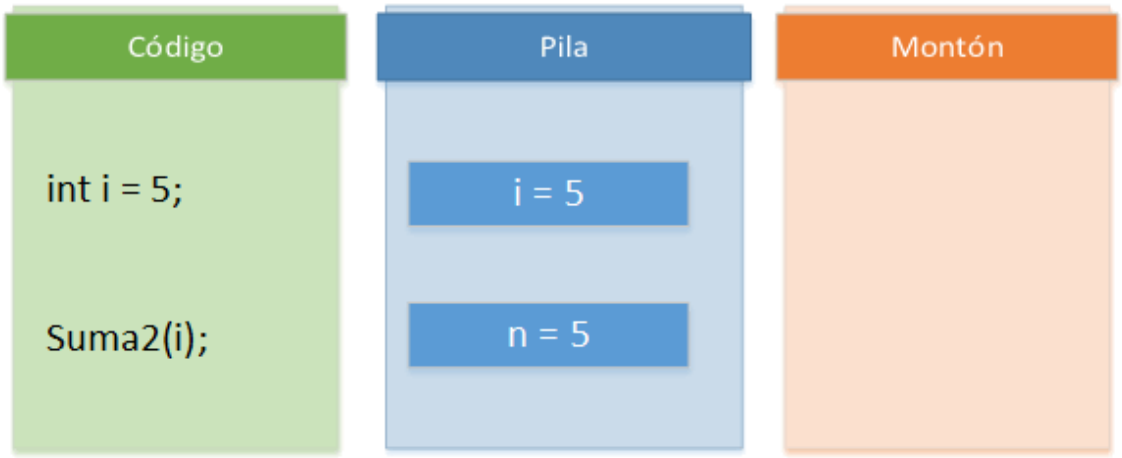

Es decir, **los cambios son apreciables en el origen**, fuera de la propia función.

## ¿A qué es debido esto?

El motivo es que, como hemos dicho, los tipos por referencia se almacenan siempre en el montón, y lo que se pasa a la función como parámetro no es una copia del dato, como en el caso de los tipos por valor, sino **una copia de la referencia al dato.**

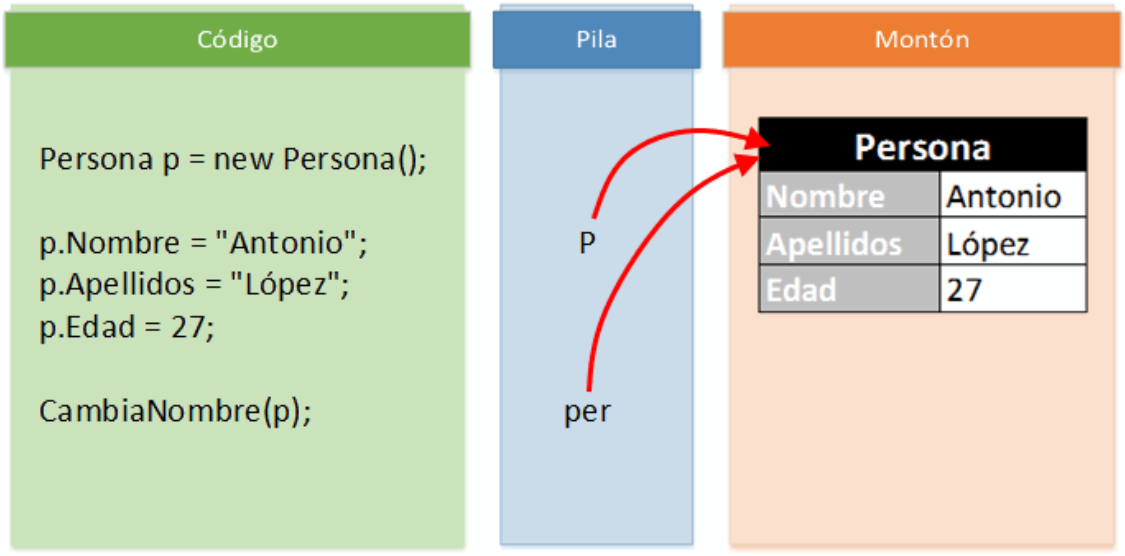
Esto que suena muy lioso es fácil de entender si lo visualizamos.

Vamos a ver en un gráfico la situación de la pila y el montón en el caso del primer código, donde solo manejábamos un tipo por valor:




Como podemos observar, al definir la variable **`i`** ésta se almacena en la pila directamente ya que es un tipo por valor. Al llamar a la función **`Suma2`** pasándole "**`i`**" como argumento, lo que ocurre es que el valor de la variable **`i`** se copia a la pila y se asigna al parámetro **`n`** de la función (repasa el código anterior). De esta forma tenemos dos copias del dato en la pila, por eso al modificar **`n`** no se afecta para nada al valor original de **`i`**. En el montón, por cierto, no tenemos nada almacenado.

Consideremos ahora el código del segundo ejemplo y veamos qué se genera en la pila y en el montón en este caso:




Ahora la cosa cambia bastante. Al declarar una variable de tipo **`Persona`** se genera una instancia de la misma en el montón, ya que es un tipo por referencia. Al asignar los valores de las diferentes propiedades, esos datos se almacenan en el montón asociadas a la instancia de la clase **`Persona`** que acabamos de crear. **Lo que realmente almacena la variable `p` es una referencia a los datos** de esa persona que están en el montón.

> Es decir, ahora en la pila no se almacenan los datos en sí, sino tan solo un "puntero" a los mismos. Es una forma indirecta de referirse a ellos, no como antes en los tipos por valor que el dato se almacenaba directamente en la pila.

Al llamar a la función **`CambiaNombre`**, en la variable local **`per`** que constituye el parámetro de la función, **lo que se duplica ahora en la pila es la referencia al objeto**, no el objeto en sí. Es decir, al hacer la llamada disponemos de dos variables que apuntan al mismo tiempo a los datos almacenados en el montón. Por eso cuando cambiamos una propiedad del objeto, al estar ambas referencias apuntando a los mismos datos, los cambios se ven desde los dos lados.

> En realidad el comportamiento de la pila es idéntico en ambos casos, lo que cambia es la información que se almacena, que es el propio dato en el caso de los tipos por valor, y la referencia al dato en el caso de los tipos por referencia.

Este modo de funcionamiento es la diferencia fundamental que existe entre los tipos por valor y los tipos por referencia, y es muy importante tenerla clara pues tiene muchas implicaciones a la hora de escribir código.

# Boxing y UnBoxing de variables

por **José Manuel Alarcón**

Un efecto importante a tener en cuenta a la hora de usar los tipos por valor, es el conocido como **Boxing**. Éste consiste en que el compilador se ve forzado a convertir un tipo por valor en un tipo por referencia, y por tanto a almacenarlo en el montón en lugar de en la pila.

¿Por qué puede ocurrir esto?

Bueno, pues por muchos motivos, pero los más comunes son dos:

## 1.- Conversión forzada a tipo por referencia

Que hagamos una conversión forzada de un tipo por valor en uno por referencia en nuestro código. Por ejemplo:

```
int num = 1;
object obj = num;
```

De acuerdo, no es un código muy útil ni muy razonable, pero sirve para ilustrar la idea. Lo que hacemos es asignar un tipo por valor a una variable pensada para albergar un tipo por referencia compatible. En este caso **`object`** es la clase base de todas las demás, así que se le puede asignar cualquier cosa.

En este ejemplo dado que **`num`** es un tipo por valor y **`obj`** es un tipo por referencia, para poder hacer la asignación el compilador se ve forzado a convertir uno en el otro. Es decir, en "envolver" al tipo básico en un tipo por referencia y colocarlo en el montón. Lo que tendríamos al final es un nuevo objeto en el montón que albergaría el valor original (1 en este caso), y en la pila habría una referencia a este objeto.

A esta operación se le llama operación de ***Boxing***, porque es como si envolviésemos al pobre valor en una caja para poder llevarlo cuidadosamente al montón y referenciarlo desde la pila.

A la operación contraria se le llama ***UnBoxing***, y se produce cuando el tipo por referencia se convierte o se asigna a un tipo por valor de nuevo. Siguiendo con el código anterior, forzaríamos un unboxing con una línea así:

```
int num2 = (int) o;
```

Ahora, al forzar la conversión del objeto (tipo por referencia) en un tipo por valor (un número entero), provocamos el deshacer lo anterior.

¿Para qué querríamos hacer algo como esto? Pues por ejemplo porque tenemos que llamar a una función o clase que acepta objetos como argumentos, pero necesitamos pasarle tipos por valor.

Hoy en día como existen los genéricos no es tan habitual, pero anteriormente había muchas clases especializadas que podían trabajar con varios tipos de datos diferentes y solo aceptaban **`object`** como argumento (mira la colección ArrayList, por ejemplo).

## 2.- Usar un tipo por valor como si fuera por referencia

Este tipo de *boxing* es más sutil y puede pasar inadvertido a muchos programadores.

Como sabemos todos los objetos de .NET heredan de la clase **`Object`**. Esta clase ofrece una serie de métodos básicos que todos los objetos de .NET tienen, ya que heredan de ésta. Así, por ejemplo, el método **`ToString()`** está definido en **`Object`** y permite obtener una representación en formato texto de un objeto cualquiera. Además dispone también de un método **`GetType()`** que nos devuelve el tipo del objeto actual.

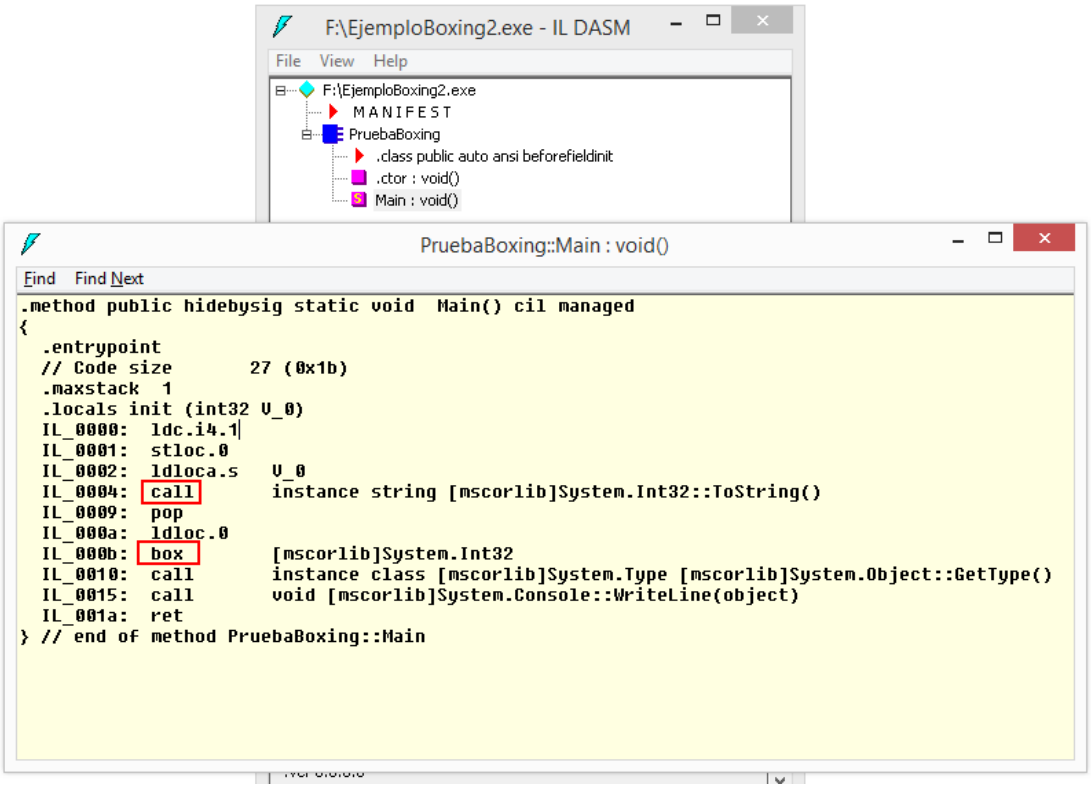
Por lo tanto podemos escribir algo como esto sin ningún problema:

```
int num = 1;
string s = num.ToString();
Console.WriteLine(num.GetType());
```

O sea, llamamos a **`ToString()`** para obtener el número como una cadena, y también llamamos a **`GetType()`** para, en este caso, mostrar por la consola el tipo del número (que es "System.Int32", que es la estructura que lo representa).

Al ejecutar un código tan sencillo como este, veamos qué ocurre por debajo. Para ello lo que he hecho es compilarlo y descompilarlo usando la herramienta **`ILDASM.exe`** (que encontrarás en **`C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools`** si tienes instalado Visual Studio).

Este es el aspecto del programa anterior en lenguaje intermedio:




Aquí debemos fijarnos en las dos instrucciones recuadradas en la figura anterior.

La primera se corresponde a la llamada a **`ToString()`**, y la segunda a la llamada a **`GetType()`**. Fijémonos primero en esta última.

La instrucción de bajo nivel que se utiliza se llama, mira tú por donde, **box**. Es decir, antes de hacer la llamada a **`GetType()`** se está convirtiendo el número en un tipo por valor, y luego en la siguiente línea ya se llama a la función que buscábamos en la clase base **`Object`**. Esto es una operación de *boxing* igual a la que vimos en el caso anterior.

Sin embargo en la primera llamada a **`ToString()`** se realiza la llamada de la manera normal y no ocurre *boxing* alguno. ¿Por qué?

Pues porque da la casualidad de que la estructura **`System.Int32`** que es la que alberga a los números enteros, tiene redefinido el método **`ToString()`** por lo que se puede llamar directamente desde el tipo por valor. Sin embargo no redefine **`GetType()`** por lo que para poder usarlo se debe convertir primero en un objeto mediante la operación de *boxing* y luego llamar al método.

Como vemos, dos líneas aparentemente iguales por debajo actúan de una manera muy diferente.

## Sutilezas con las que podemos evitar el boxing

Un caso muy común de *boxing* debido a esta causa es cuando utilizamos el método **`String.Format()`** para crear una cadena a partir varios datos, o lo que es casi lo mismo, el método **`Console.WriteLine()`**.

Por ejemplo, si escribimos algo como esto:

```
Console.WriteLine("El valor de mi variable es: {1}", num);
```

Estaremos provocando un *boxing*, ya que **`WriteLine`** espera como parámetros después de la cadena de formato, objetos genéricos. Así que para llamar al método primero se convierte num en un objeto y luego se le pasa.

Sin embargo si escribimos esto otro, muy parecido:

```
Console.WriteLine("El valor de mi variable es: {1}",
num.ToString());
```

En este caso no existirá *boxing*, ya que ahora no le estamos pasando un tipo por valor, sino un tipo por referencia (una cadena). El resultado es exactamente el mismo, pero gracias a esto hemos evitado hacer el boxing y hemos mejorado el rendimiento

(imperceptiblemente en este caso, pero…). Además como hemos visto hace un momento el **`ToString()`** no provoca tampoco *boxing* por estar definido en **`Int32`**.

## ¿Qué importancia tiene todo esto?

Las operaciones de *boxing* y *unboxing* tienen un impacto en el rendimiento y en la gestión de la memoria. Como ya sabemos, el manejo de valores en la pila es más rápido y eficiente que en el montón, y además la propia operación de envolver el tipo por valor para meterlo en el montón y luego referenciarla desde la pila es costosa también.

La importancia de este efecto en aplicaciones normales es imperceptible. Sin embargo, en aplicaciones donde el rendimiento es crucial y se realizan potencialmente muchos millones de operaciones de *boxing* y/o *unboxing*, tener el concepto claro y evitarlo es un algo muy importante y puede impactar mucho en el rendimiento.

En cualquier caso, es importante conocer el concepto y saber identificarlo cuando sea necesario.

# Clases y estructuras en .Net: cuándo usar cuál

por **José Manuel Alarcón**

Una de las cuestiones de concepto más comunes entre los alumnos de cursos de fundamentos de programación con .NET es **la diferencia que existe entre clases y estructuras**.

Al fin y al cabo se parecen mucho: ambas se pueden definir de manera similar, disponen de métodos y propiedades, ambas se pueden instanciar...

Por ejemplo, si definimos en C# una clase de nombre **`Punto`** como esta:

```
public class Punto
{
    public int x,y;
    public Punto(int X, int Y)
    {
        this.x = X;
        this.y = Y;
    }

    public override string ToString()
    {
        return string.Format("Coordenadas del punto
({0},{1})", this.x, this.y);
    }
}
```

Podemos muy fácilmente convertirla en una estructura cambiando la palabra clave **`class`** por **`struct`**:

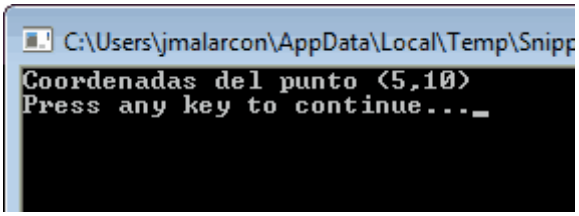
```
public struct Punto
{
    public int x,y;
    public Punto(int X, int Y)
    {
        this.x = X;
        this.y = Y;
    }

    public override string ToString()
    {
        return string.Format("Coordenadas del punto
({0},{1})", this.x, this.y);
    }
}
```

Es exactamente igual, salvo esa palabra clave: tiene un par de campos (*x* e *y*), un constructor y redefine el método **`ToString`** de la clase base **`Object`**.

La definamos como clase o como estructura la manera de usarla es exactamente la misma:

```
Punto punto1 = new Punto(5,10);
Console.WriteLine(punto1);
```

Es decir, se instancia con dos valores por defecto, y se muestra por pantalla (en este caso en la línea de comandos), obteniendo esto:




Entonces, si aparentemente funcionan igual, **¿cuál es la principal diferencia entre una estructura y una clase?**

La principal diferencia entre una estructura y una clase en .NET es que **las primeras son** tipos por valor y las otras tipos por referencia. Es decir, **aunque las estructuras pueden trabajar como clases, realmente son valores ubicados en la pila directamente**, y no referencias a la información en memoria.

Esto, tan teórico, significa, como hemos visto antes, que **las estructuras se pueden gestionar más eficientemente al instanciarlas** (se hace más rápido), sobre todo en grandes cantidades, como una matriz. Al crear una matriz de estructuras, éstas se crean consecutivamente en memoria y no es necesario instanciar una a una y guardar sus referencias en la matriz, por lo que es mucho más rápido su uso.

Además, si pasas una estructura como parámetro a un método, al hacerlo se crea una copia de la misma en lugar de pasar una referencia (como ya hemos visto), y por lo tanto los cambios aplicados sobre ella se pierden al terminar de ejecutarse la función. Es decir, si cambias el valor de una propiedad dentro del código de la función, este cambio no se verá desde el código que llama a la función.

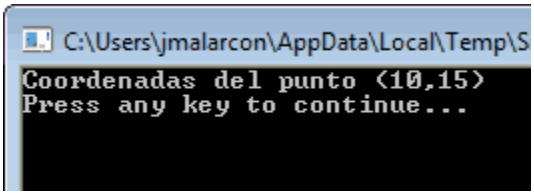
Por ejemplo, si definimos esta función que toma un punto y le suma una determinada cantidad a sus coordenadas:

```
public void SumaCoordenadas(Punto p, int incremento)
{
    p.x += incremento;
    p.y += incremento;
}
```

Al llamarlo de esta manera:

```
Punto punto1 = new Punto(5,10);
SumaCoordenadas(punto1, 5);
Console.WriteLine(punto1);
```

Lo que obtenemos por pantalla cambia totalmente si **Punto** es una clase o una estructura. Si se trata de una clase, lo que recibe la función es una referencia a la instancia de la clase, es decir, **se obtiene acceso a la verdadera clase y la información que maneja**. Por ello, cuando dentro de la función se cambian los valores de *x* e *y*, se están cambiando los verdaderos valores de esas propiedades. Es decir, dentro de la función el parámetro **p** apunta a la mismo "almacén" de datos en memoria que **punto1**. Por lo tanto, al mostrar por consola sus valores tras llamar a la función **SumaCoordenadas** lo que obtenemos es:




O sea, se ha modificado los valores de **x** e **y** en el objeto **punto1**.

Sin embargo **ese mismo código exactamente aplicado sobre Punto cuando lo hemos definido como una estructura funciona de manera totalmente distinta**. En ese caso lo que obtenemos en **p** dentro de la función no es una referencia a la misma memoria a la que apunta **punto1**, sino que obtenemos una copia de la información. Por ello, aunque modifiquemos **x** e **y** dentro de la función, al terminar de ejecutarse ésta esos valores se pierden, puesto que ambas variables apuntan a objetos diferentes en la memoria. La ejecución del código anterior cuando **Punto** es una estructura devuelve lo mismo que en la primera figura:

```
Coordenadas del punto (5,10)
```

## ¿Cuándo es interesante usar una estructura en lugar de una clase?

**Lo habitual es no usar estructuras casi nunca**, salvo para realizar optimizaciones.

Un caso concreto de uso de una estructura sería si, por ejemplo, quieres trabajar con números complejos, que, como seguramente sabrás, están representados por dos cifras: una parte real y una imaginaria. Si quieres tratarlos como un número más, es decir, que sean un tipo por valor, lo lógico es que los hagas estructuras para no tener que instanciarlos y para que cuando llames a métodos pasándole números complejos éstos se pasen por valor, y no se modifiquen con lo que hagas dentro del método al que se los pasas. A todos los efectos se comportarían como si fueran otro tipo por valor, como un entero o un *double*.

Otro ejemplo, más común, es cuando debes trabajar con una cantidad muy elevada de objetos en memoria, uno tras otro. Por ejemplo, imagínate que tienes que hacer una transformación sobre puntos tridimensionales en el espacio. Te puedes crear una clase que los represente, pero si debes ir creando y procesando miles de ellos será mucho más eficiente crearlos usando estructuras porque de este modo se crean en la pila y se libera su espacio en cuanto dejas de necesitarlos (los objetos quedarían en memoria hasta la próxima vez que los necesites).

También hay escenarios de interoperabilidad, cuando se llama a código nativo de Windows usando **`PInvoke`**, o a objetos **`COM`** usando **`COM/Interop`**, donde es necesario pasar estructuras y no clases.

Pero vamos, como resumen quédate con que, salvo casos raros, lo habitual es que utilices clases en tu código.

# Diferencias al usar Clases y Estructuras en matrices

por **José Manuel Alarcón**

Existe una cuestión consecuencia de las diferencias entre clases y estructuras que a muchos programadores les suele pasar inadvertida. Se trata del comportamiento obtenido al instanciar una matriz.

Imaginemos que tenemos una sencilla estructura como esta:

```
struct Usuario
{
   public string Nombre;
   public string Apellidos;
}
```

Si creamos una matriz de estructuras de este tipo así:

```
Usuario[] usuarios = new Usuario[10];
```

Podremos inmediatamente asignar valores a cada elemento de la matriz, ya que la estructura al ser un tipo por valor está disponible para que la usemos:

```
usuarios[0].Nombre = "Pepe"; //No fallará
```

Sin embargo si cambiamos la definición de Usuario convirtiéndola en una clase:

```
class Usuario
{
   public string Nombre;
   public string Apellidos;
}
```

El código de asignación anterior fallará debido a que al crear la matriz (el código es exactamente el mismo) no se crean al mismo tiempo nuevos objetos de la clase **`Usuario`**, sino que la matriz contiene **referencias a objetos** de tipo **`Usuario`**, pero no objetos de tipo **`Usuario`**, cosa que sí pasa al instanciar tipos por valor como las estructuras.

Para hacer la asignación primero habrá que instanciar un objeto y asignarlo al elemento de la matriz:

```
//Esto fallaría
//usuarios[0].Nombre = "Pepe";


//Esto es lo correcto en este caso
Usuario u = new Usuario();
u.Nombre = "Pepe";
usuarios[0] = u;


//O bien esto otro que es equivalente...
Usuario u = new Usuario();
usuarios[0] = u;
usuarios[0].Nombre = "Pepe";
```

Esta sutil diferencia **hace mucho más fácil crear y usar matrices de estructuras que matrices de clases**, aunque claro, no todo son ventajas. Las matrices de estructuras

ocupan más memoria y el hecho de utilizarlas después puede implicar realizar más procesos de *Boxing*/*Unboxing* que son costosos en términos de rendimiento. Aun así, en la mayor parte de las aplicaciones este hecho no nos afectará.

En cada caso hay que sopesar qué es lo mejor pero conviene ser consciente de estas diferencias, que por cierto no existen (en lo que se refiere a la forma de usarlas en código) en la mayor parte de las demás situaciones.

# ¿Qué diferencia hay entre usar o no "New" al declarar una variable cuyo tipo es una estructura?

por **José Manuel Alarcón**

Consideremos por ejemplo esta estructura en C#:

```
public struct Punto
{
    public int x, y;
}
```

Puedes declarar una variable que las use de dos formas diferentes: usando New o no.

```
Punto p;
Punto p = new Punto();
```

## ¿Cuál es la diferencia?

A priori parece no haber ninguna. Sin embargo sí que la hay, y tiene que ver con **extender las estructuras con propiedades y métodos**.

A las estructuras se les puede añadir métodos, propiedades, etc... como si fueran clases. Por ejemplo, consideremos el siguiente código en C#:

```
public struct Punto
{
    int x, y;

    public int X
    {
        get { return x; }
        set { x = value; }
    }

    public int Y
    {
        get { return y; }
        set { y = value; }
    }
}
```

Si escribimos el siguiente código (el más habitual):

```
Punto p;
Console.WriteLine (p.X);
```

Obtendremos un error que nos dice que la variable interna '**x**' **no está inicializada**. Sin embargo, con este otro código:

```
Punto p = new Punto();
Console.WriteLine (p.x);
```

No se produciría error alguno.

Lo importante aquí es que, **al ser declaradas por defecto** (sin constructor), las estructuras **no inicializan sus miembros internos**, por lo tanto **antes de usarlas debemos asignarles un valor.**

Cada propiedad tiene un campo interno que no está inicializado cuando intentamos usarlo a través de la propiedad por primera vez, y obtendrías un error al hacerlo (no se sabe si tu propiedad antes de asignarlo intenta hacer algo con él y por lo tanto te obliga a tenerlo inicializado).

Si usas el constructor por defecto explícitamente (con la palabra clave **new**) lo que éste hace es **inicializar todos los miembros internos** a su valor por defecto, y a partir de ese momento ya te deja utilizarlos. Si no usas **new** no se llama al constructor por defecto y los campos están sin inicializar.

# Palabras clave ref y out para paso de parámetros por referencia

por **José Manuel Alarcón**

Como ya hemos visto antes, cuando se pasa un tipo por valor, éste siempre se pasa por valor (por supuesto) y no se ven sus modificaciones fuera del ámbito de la función. Y cuando se pasa un objeto como parámetro a cualquier método de .NET, éste se pasa **siempre** por referencia.

Sin embargo, existe una palabra clave en el lenguaje llamada `ref` que, según la documentación, sirve para pasar parámetros por referencia. ¿Cuál es su objeto entonces si los objetos ya se pasan por referencia siempre?

Pues su propósito es el de **poder pasar por referencia tipos por valor**.

Consideremos el siguiente ejemplo similar a uno visto anteriormente:

```
class Persona
{
    public string Nombre = "Original";
}

....
void CambiaNombre(Persona p)
{
    p.Nombre = "Cambiado";
}

...
Persona per = new Persona();
CambiaNombre(per);
Console.WriteLine(per.Nombre);
```

En este fragmento, al llamar a la función **`CambiaNombre`** pasándole un objeto de la clase **`Persona`**, lo que obtenemos en la consola es la palabra "*Cambiado*". Esto se debe a que el objeto se pasa por referencia. Hasta aquí todo normal.

Si cambiamos la definición de la clase **Persona** y la convertimos en una estructura (que, para entendernos, es igual que una clase pero es un tipo por valor y por tanto se almacena siempre en la pila):

```
struct Persona
{
    public string Nombre;
}
```

Al ejecutar el mismo código de antes en la consola veríamos la palabra "*Original*", ya que al ser una estructura un tipo por valor, por lo tanto lo que se pasa como parámetro a la función es una copia del objeto, no una referencia al objeto original.

La forma que proporciona el lenguaje C# para forzar el paso a las funciones de referencias incluso cuando trabajamos con tipos por valor es la palabra clave **ref**. Al colocarla delante del parámetro el compilador envía una referencia a la estructura y no una copia, por lo que se conservan las modificaciones hechas dentro del método. En nuestro ejemplo bastaría con poner:

```
void CambiaNombre(ref Persona p)
```

y al llamar al método también:

```
CambiaNombre(ref per);
```

El problema de esto es que antes de pasar el parámetro hay que convertir el tipo por valor en un tipo por referencia, lo que se conoce técnicamente como operación de ***Boxing***. Y al terminar la llamada se debe hacer un ***Unboxing*** que es la operación contraria. Dicho proceso, como sabemos, es más costoso que simplemente pasar una referencia, por lo que el rendimiento de la aplicación puede bajar si se hace con mucha frecuencia (dentro de un bucle largo, por ejemplo). Además, el propósito perseguido

normalmente cuando se crea una estructura es precisamente usarla como tipo por valor, por lo que en la práctica la palabra clave **`ref`** se usa más bien poco.

Existe una palabra similar llamada `out` que también se puede aplicar a un parámetro para poder obtener referencias a objetos modificados, y que tiene sentido incluso con clases.

La diferencia entre **`ref`** y **`out`** es que, en el primer caso, el objeto que se pase al parámetro tiene que estar inicializado (es decir, tiene que ser una instancia concreta de un tipo por valor) mientras que en el segundo caso puede ser simplemente una variable sin inicializar que a la vuelta de la función contendrá una información concreta como valor de retorno.

# Tipos genéricos en .NET y C#

por **José Manuel Alarcón**

La genericidad en .NET, y en particular en C#, permite crear código que es capaz de trabajar de manera eficiente con varios tipos de datos al mismo tiempo. Es decir, podremos escribir una función o una clase que actúe como patrón genérico de una forma de trabajar, y que luego se puede particularizar para usarlo con tipos de datos concretos cuando sea necesario.

> Ello permite crear código de métodos, estructuras, clases e incluso interfaces sin preocuparnos por los tipos de datos que vamos a utilizar a la hora de la verdad.

Antes de la existencia de los tipos genéricos, cuando queríamos escribir código que pudiese trabajar indistintamente con dos o más tipos de datos no nos quedaba más remedio que utilizar el tipo **`object`** que es la raíz de todas las clases, o hacer sobrecargas de la misma función repitiendo el mismo código tan solo por que trabajaba con un tipo diferente. Con los *Generics* de .NET esto ya no es necesario, y nuestro código puede disfrutar de la robustez de tipos concretos sin tener que comprometerse por adelantado con ninguno.

El ejemplo clásico de este concepto es el de la clase *Pila* (*Stack* en inglés). La clase **`Stack`** se usa para almacenar objetos o tipos básicos de forma LIFO (*Last In First Out*), es decir, se almacenan elementos y luego se pueden recuperar empezando siempre por el último que se haya introducido. Independientemente del tipo de objetos que se almacenen en la pila, el código para gestionarlos es el mismo. La solución clásica sería usar **`object`** o bien crear diversas versiones sobrecargadas de los métodos de la pila que tomasen los diferentes tipos de objeto que ésta puede manejar. Algo muy tedioso y poco eficiente. Si todos los objetos que se almacenarán son del mismo tipo es una ineficiencia utilizar el tipo **`object`**, siendo mejor usar tipos concretos.

En C#,  por ejemplo, la clase pila se definiría de esta manera:

```
public class Stack<CualquierTipo>
{
    private CualquierTipo[] m_items;
    public CualquierTipo Pop() {.....}
    public void Push(CualquierTipo data) {....}
}
```

Fíjate en que si, por ejemplo, la clase la estuviésemos definiendo para ser utilizada con números enteros, la definición sería la misma, solo que, en donde ahora hemos puesto **`CualquierTipo`** pondría **`int`**. Es la única diferencia, ya que lo demás sería idéntico.

Con los genéricos podemos ahorrarnos por el momento indicar el tipo de dato a utilizar y lo sustituimos por un *"comodín"* que va entre los símbolos **<** y **>**, como acabamos de ver.

Ahora, para utilizar nuestra pila genérica con un tipo de datos concreto, debemos hacerla concreta. Esto es, debemos crear una instancia de la misma diciendo qué tipo de datos queremos utilizar con ella. En código esto se traduce en que, por ejemplo, si queremos crear una pila de números enteros, tendríamos que escribir:

```
Stack<int> pila = new Stack(<int>);
pila.Push(5);
pila.Push(1);
int x = pila.Pop();
```

Como vemos se declara la pila indicando entre signos de menor y mayor el tipo que queremos usar para sustituir al genérico. Luego la podemos usar directamente como si siempre hubiera estado definida con ese tipo de datos. Es de lo más cómodo y flexible.

Por supuesto se puede usar cualquier tipo a la hora de definir una instancia concreta. Por ejemplo, si tenemos una clase **`Usuario`** y queremos gestionar una pila de usuarios sólo tenemos que crear una instancia específica así:

```
Stack<Usuario> pila = new Stack(<Usuario>);
```

También es posible utilizar varios tipos genéricos en una definición, sólo hay que separarlos con comas dentro de la pareja **<** y **>**.

# Cómo hacer una función genérica para conversión entre tipos

por **José Manuel Alarcón**

La plataforma .NET permite crear métodos y clases genéricas que trabajen de la misma manera, independientemente del tipo que utilicen para la operación.

Por ejemplo, ¿cómo podríamos crear una función que sirviera para transformar de forma genérica entre dos tipos de datos cualesquiera?. Es decir, una a la que le pasases una variable de un valor o una referencia de un determinado tipo y que devolviera el mismo dato pero con un tipo diferente.

En realidad ya existe una función similar desde la primera versión de .NET, y es el método **ChangeType** de la clase **Convert**. Este método tiene varias sobrecargas pero la que se usa más habitualmente toma como primer parámetro un valor a convertir y como segundo una definición de tipo al que deseamos convertir el anterior. Así, por ejemplo, para cambiar desde un **double** a un entero, escribiríamos algo así:

```
double d = 12;
int i = (int) Convert.ChangeType(d, typeof(int));
```

Esto está bien, pero es bastante lioso. Para empezar el método devuelve un objeto que hay que convertir en su tipo adecuado subyacente, lo que a fin de cuentas nos resta genericidad pues ¿cómo encapsulas en un método lo anterior? Además la sintaxis es liosa y difícil de leer.

¿No sería mejor disponer de un método más sencillo y directo de conseguir lo mismo?

Es aquí donde entran los genéricos. Pensemos en utilizar un método que use genericidad para conseguir el mismo efecto pero que sea sencillo de utilizar desde código y funcione de un modo realmente general. El resultado es este código:

```
public static T Convertir<T>(object v)
{
   try
   {
      return (T) Convert.ChangeType(v, typeof(T));
   }
   catch
   {
      return (T) Activator.CreateInstance(typeof(T));
   }
}
```

Con éste la conversión entre dos tipos cualquiera es tan fácil como en este ejemplo:

```
DateTime d = DateTime.Now;
string s = Convertir<string>(d);
```

La sintaxis es mucho más clara y directa ¿no?

Realmente lo único que hace es encapsular el uso del método **ChangeType** permitiendo su uso de forma genérica y facilitando el uso general desde el código.

Este método además tiene la particularidad de que siempre devuelve un valor del tipo adecuado ya que si la conversión falla instancia un objeto nuevo de ese tipo. Esto último no tiene por qué ser interesante siempre, así que es fácil sustituirlo por el lanzamiento de una excepción de tipo **InvalidCastException** cuando la conversión no sea posible.

# Tipos anónimos en .NET y C#

por **José Manuel Alarcón**

Los tipos anónimos nos permiten definir clases de un solo uso dinámicamente, lo cual tiene una importancia vital en el manejo de resultados de consultas LINQ. De hecho, se añadieron al lenguaje precisamente para ello.

Una clase anónima se define con la palabra clave **new**, igual que una clase normal, pero no se le da nombre (de ahí lo de anónima) y se crea sobre la marcha a partir de los datos que se le pasen al constructor. Por ejemplo:

```
var nuevoMVP = new { Nombre = "José Manuel Alarcón",
Especialidad = "ASP.NET", PrimerAño = 2004 };
```

Ahora ya podemos usar la variable **nuevoMVP** como si hubiésemos instanciado una clase definida en el código de manera normal, por ejemplo:

```
Console.Write(nuevoMVP.Nombre);
```

Esto permite manejar datos *sintetizados* dinámicamente como resultados de consultas LINQ que contengan campos arbitrarios obtenidos desde la fuente de datos subyacente.

Como puedes suponer, realmente no son anónimos. Si usas reflexión (o directamente desensamblas un código que los use) verás que el compilador genera internamente una clase con un nombre *raro*, lo que pasa que para el uso que hacemos es completamente irrelevante.

Otro detalle es que, en realidad, se denominan tipos anónimos **inmutables**. Ello se debe a que no se pueden modificar sus valores (las propiedades creadas son de solo lectura). Como vemos su uso es muy específico y relacionado con lo que comentaba de LINQ.

Por cierto, ¿te has fijado en la forma tan *extraña* de definir la variable que contiene la referencia al tipo anónimo? ¿Qué es eso de **var**? Lo veremos enseguida. Pero antes…

# Métodos anónimos en C#

por **José Manuel Alarcón**

Los ***métodos anónimos*** están pensados para simplificar la definición de manejadores de eventos cuando las tareas a realizar son simples.

Imaginemos que queremos asociar una única línea de código (o dos) como acción a realizar cuando se produzca un evento. Por ejemplo, que cuando se pulse un botón se muestre un mensaje de saludo. Lo que debíamos hacer tradicionalmente era declarar una función del tipo adecuado y asociarla con un nuevo delegado al evento. Por ejemplo:

```
private void button1_Click(object sender, System.EventArgs e)
   {
        MessageBox.Show("Hola");
   }
button1.Click += new System.EventHandler(button1_Click);
```

Esto hace que al pulsar el botón se vea el saludo "Hola". Sin embargo es un poco engorroso para tan poca acción tener que crear una función como esta ¿no? Para evitarlo los métodos anónimos vienen al rescate.

Un método anónimo nos permite definir una función de manera implícita, sin necesidad de declararla. En el ejemplo anterior un método anónimo sustitutivo sería el siguiente:

```
button1.Click += delegate{ MessageBox.Show("Hola"); };
```

Raro ¿no? La única diferencia es que ahora no nos hace falta definir una función, lo hace automáticamente por nosotros el compilador a la hora de crear el ejecutable.

Dependiendo del tipo de evento que estemos asignando el compilador ya deriva de forma automática los parámetros que necesita el método anónimo así como el tipo de delegado que se debe crear (en este caso **`System.EventHandler`**). Una "virguería" vamos.

En algunos tipos de eventos, como la pulsación del ratón y otros, necesitaremos acceder a los parámetros de éste para conocer cierta información. En este caso lo único que debemos hacer es declarar los argumentos tras la palabra 'delegate', así:

```
button1.Click += delegate(object sender, EventArgs e){
MessageBox.Show(e.ToString()); };
```

Como vemos al declarar el método anónimo indicando en el delegado qué tipos de parámetro se necesitan y otorgándoles un nombre es posible hacer uso de ellos sin problemas desde el método anónimo.

# Definición automática de tipos de variables en C#

por **José Manuel Alarcón**

En C# es posible dejar al compilador la tarea de determinar el tipo de una variable que hayamos inicializado de manera explícita.

El tipo se deduce a partir de la expresión de inicialización. De este modo nos evitamos tener que declarar el tipo correcto, sobre todo en casos en los que no lo tenemos claro de antemano, como por ejemplo en consultas de LINQ, o en funciones polimórficas que devuelven diversos tipos y que todos implementan la misma interfaz.

Así, por ejemplo, en el siguiente ejemplo el compilador infiere que la variable **n** es de tipo **int** y que **s** es de tipo **string** por el tipo de los literales que se asignan a cada una:

```
var n = 5; // equivale a int n = 5;
var s = "Hola";  // equivale a string s = "Hola";
```

Por supuesto, se mantienen en vigor las reglas del lenguaje para determinar el tipo de las expresiones:

```
var m = 3 * n;  // m es de tipo int
var dosPi = 2 * Math.PI;  // dosPi es de tipo double
```

El mecanismo también funciona para otros tipos cualesquiera:

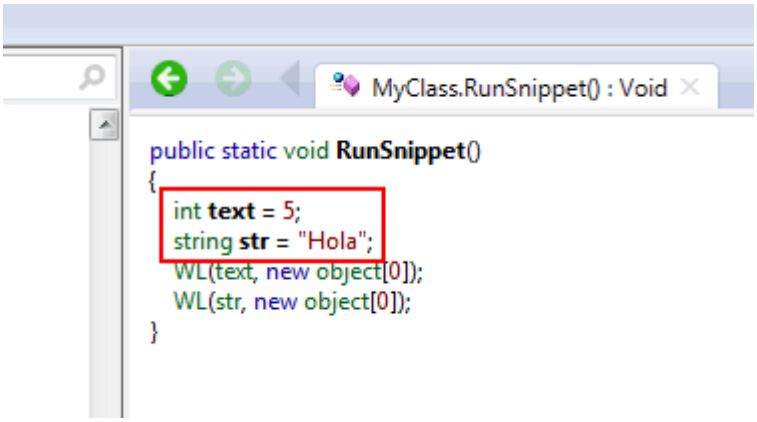
```
var p1 = new Persona("Ana", 24);  // p1 es de tipo Persona
var p2 = p1;  // p2 también
var enteros = new List();  // enteros es de tipo List
var f1 = DateTime.Now.AddMonths(1);  // f1 es de tipo DateTime
```

Los siguientes son ejemplos de usos incorrectos de **var**:

```
var p;  // la inicialización es obligatoria
var q = null;  // imposible inferir el tipo de q
var z1 = 3, z2 = 5; // no se admiten declaraciones múltiples
var r = { 1, 2, 3 }; // un inicializador de array no está
permitido
```

Es importante destacar que **esta característica no tiene nada que ver con la que ofrecen algunos lenguajes dinámicos** (como JavaScript), en los que una variable puede almacenar valores de distintos tipos a lo largo de su tiempo de vida. En este caso el compilador simplemente nos libera de la necesidad de especificar el tipo de una variable inicializada e infiere el tipo de la variable declarada mediante el *identificador especial* **var** a partir del tipo de la expresión que se le asigna.

La verificación estricta de tipos, uno de los pilares del lenguaje, sigue por supuesto en vigor, y la mejor manera de comprobarlo es utilizando una herramienta como Reflector o DotPeek, que nos dejarán ver claramente que esas variables tienen un tipo concreto asignado por el compilador:




En este caso se muestra cómo el compilador detecta el tipo correcto para cada variable en el primero de los ejemplos, más arriba.

La declaración implícita del tipo de una variable puede utilizarse también en otras construcciones del lenguaje, como el bucle **`foreach`** o la sentencia **`using`**:

```
foreach (var k in listaEnteros)  // k es de tipo int
    Console.WriteLine(k);

using (var con = new SqlConnection(cadenaConexion))
    // con es de tipo SqlConnection
{
    con.Open();
    // interactuar con la base de datos...
}
```

Mucha gente tiene miedo de utilizar esta construcción por creer que va a perder la comprobación estricta de tipos que ofrece el lenguaje. Como has podido ver, nada más lejos de la realidad, así que anímate a usarlo.

# Tipos anulables en C#

por **José Manuel Alarcón**

Una interesante característica que ofrece la plataforma.NET es la existencia de *Tipos anulables* o, en inglés, *Nullable types*.

Se trata de un tipo especial de dato que permite que los tipos base por valor se puedan comportar como valores por referencia nulos cuando sea necesario.

De este modo este nuevo tipo anulable **permite representar valores concretos o la ausencia de valor**. Por ejemplo, una variable de tipo **`int`** siempre contiene un número entero. Aunque no le asignemos explícitamente nada, al ser un tipo base por valor siempre toma un valor por defecto, en este caso un 0. Pero claro, un 0 no es lo mismo que decir "esta variable no contiene valor alguno".

En el caso de las clases (tipos por referencia) la ausencia de *valor* se representa asignándole un nulo a la variable, pero en los tipos por valor esto no es posible.

¿No sería estupendo disponer de un híbrido entre estos dos comportamientos?

Pues sí. Veamos el porqué...

El caso prototípico en el que esto es muy útil es al obtener ciertos campos desde una base de datos, los cuales contienen tipos base (enteros, cadenas, etc...) pero que **permiten albergar valores nulos**.

Este tipo de datos pueden estar definidos en la base de datos (en cuyo caso es un valor concreto) o ser nulos (o sea, no existen). Sin embargo, al intentar asignar un valor nulo (o también un **`System.DBNull`**) a un tipo por valor como un entero se producirá una excepción. Esto es lo que pasaba en las versiones antiguas de .NET. Pero ya no gracias a los tipos anulables. Vamos a verlo...

## ¿Cómo se definen?

Para definir un tipo anulable en C# se usa una interrogación al final de la declaración del tipo. Así por ejemplo:

```
int? miEnteroAnulable;
```

Es decir, se declara igual que un entero normal pero con una interrogación.

## ¿Es una onda o es una partícula? ¿Cómo se comporta?

Cuando declaramos un tipo como *Nullable*, cuando este contiene un valor se comporta a todos los efectos como un tipo por valor, o sea, como siempre. Cuando se le asigna un nulo no se queja (no hay excepción) y, en realidad, se convierte en un tipo por referencia para permitirlo mediante una operación de *Boxing*. Es como la luz que es dos cosas al mismo tiempo, onda y partícula ;-)

Lo que pasa por debajo es que la CLR trata el tipo anulable como una estructura (por lo tanto, como un tipo por valor) cuando éste se comporta como un valor, y como una clase (o sea, un tipo por referencia) cuando se comporta como una referencia. Esto lo hace de modo transparente para nosotros.

El tipo *Nullable* dispone de dos propiedades fundamentales que nos permiten lidiar con él de forma idéntica, bien sea una clase o bien una estructura:

- La propiedad **`HasValue`** nos permite saber si el tipo contiene un valor o por el contrario es nulo.
- La propiedad **`Value`** nos devuelve el valor subyacente del tipo (un número entero o lo que sea) siempre y cuando **`HasValue`** sea verdadera. Si intentamos leer esta propiedad cuando **`HasValue`** es falsa se genera una excepción de tipo *InvalidOperationException*.

## En resumen...

Los tipos anulables tienen grandes aplicaciones, sobre todo en el mundo de las bases de datos y están integrados en la propia CLR por lo que se pueden usar desde cualquier lenguaje. Aunque tras haber leído lo anterior te puedan parecer alejados de tus necesidades actuales, dales una oportunidad. Una vez que empieces a utilizarlos no podrá vivir sin ellos :-)

# El operador doble interrogación en C#

por **José Manuel Alarcón**

Los **tipos anulables** en .NET son una interesante característica a la que se le puede sacar bastante partido, como hemos visto.

Dado que los tipos anulables pueden contener un valor o un nulo, debemos realizar continuamente comprobaciones en el código para ver si tenemos nulos o no. Por ejemplo, algo así:

```
int? num1 = 5; //En la realidad lo obtendríamos de una BD o
de otro sistema externo
int num2;
if (num1 == null)
   num2 = 0;
else
   num2 = num1;
```

Lo cual es sencillo pero es tedioso de escribir si hay que hacerlo continuamente. Claro que podemos reducir el código usando el **operador ?** de toda la vida de C#:

```
int? num1 = 5;
int num2 = (num1==null)?0:num1;
```

Es idéntico, más reducido, pero tampoco es que mejore mucho la legibilidad ¿verdad? Bien, C# nos ofrece el **operador ??** que hace exactamente lo mismo que lo anterior y tiene las virtudes de ser conciso y a la vez legible (si sabes qué significa la doble interrogación, claro). Lo anterior quedaría con este operador así:

```
int? num1 = null;
int num2 = num1??0;
```

Es fácil de leer si lo interpretas así: asignar a num2 el valor de num1 o un cero si es nulo.

# Propiedades automáticas en C#

por **José Manuel Alarcón**

La **implementación automática de propiedades** es una capacidad sencilla, pero de gran utilidad, que fue incorporada en el lenguaje C# en su versión 3.0 ya hace bastantes años.

Generalmente cuando definíamos una propiedad de C# debemos declarar una variable privada para contener los valores, así como sus métodos **`set`** y **`get`** para asignarla y recuperarla. Si no necesitamos añadir lógica, y la propiedad sólo es un envoltorio de un campo privado (algo muy común), **acabamos repitiendo todo el tiempo mucho código como el mostrado a continuación**:

```
private string _nombre;

public string Nombre
{
  get
    {
      return _nombre;
    }
  set
    {
      _nombre = value;
    }
}
```

Un verdadero tedio (¡y esto solo para una propiedad!) que nos obliga a perder tiempo haciendo todo el rato lo mismo.

Con la definición automática de propiedades podemos definir una propiedad idéntica a la anterior sin esfuerzo. Basta con escribir lo siguiente:

```
public string Nombre { get; set;}
```

¡Listo!. El compilador generará todo lo demás por nosotros escogiendo un nombre aleatorio para la variable privada (bastante feo por cierto, pero que nunca se ve, pues queda en el código IL) y dejando más claro y mucho más conciso nuestro código.

# La palabra clave 'using'

por **José Manuel Alarcón**

No hay que confundirlo con el **`using`** de inclusión de un espacio de nombres que aparece en la parte de arriba de los archivos C#.

El objetivo de la cláusula using en C# es el de asegurar que los recursos asociados a un determinado objeto se liberan siempre, es decir, se emplea para asegurar que al acabar de usarlo siempre se llama al método **`Dispose()`** del objeto que referenciamos dentro de la declaración de **`using`**.

Para explicarlo de manera más sencilla, veamos su utilidad en una situación muy común, por ejemplo, un acceso a base de datos.

Supongamos el siguiente código muy habitual en ADO.NET clásico:

```
using System.Data; //Este es el using de importación de
espacio de nombres, que no es el que nos interesa
using System.Data.SqlClient;
SqlConnection conn = new SqlConnection();
try
  {
   //Aquí hacemos algo con la conexión
  }
finally
  {
   conn.Close();
  }
```

Es decir, abrimos una conexión, dentro del bloque **`try`** lanzamos una consulta o varias, etc.. El bloque **`finally`** se ha incluido para asegurarnos de que la conexión se cierra aunque se produzca un error en nuestro bloque de procesamiento, ya que de otro

modo quedaría abierta y no se podría recuperar, disminuyendo la escalabilidad de nuestro programa.

> **Nota:** De hecho, el modo adecuado de hacer esto aún sería más complicado ya que la cláusula **`finally`** que he escrito da por hecho que la conexión está abierta y la cierra, pero si ya estuviese cerrada se produciría una excepción. Así que en un código real tendríamos que comprobar antes el estado de la conexión y luego llamar a **`Close()`**. En fin, una pesadez pero la única forma buena de hacerlo.... y de hecho me consta que muchos programadores ni siquiera lo tienen en cuenta. ¡Grave error!

Para evitarnos el tedio de escribir lo anterior pero aun así cerciorarnos de que la conexión se cierra, podríamos emplear la cláusula using de la siguiente manera:

```
using System.Data; //Este es el using de importación de
espacio de nombres, que no es el que nos interesa
using System.Data.SqlClient;
using(SqlConnection conn = new SqlConnection())
  {
        //Aquí hacemos algo con la conexión
  }
```

Esto es equivalente a todo lo que comentaba en el párrafo anterior. Es decir, el **`using`** se cerciora de que, al acabar el bloque de código entre sus llaves, aunque haya un error, se llama al método **`Dispose()`** del objeto **`conn`** que hemos declarado en su bloque inicial. Este método **`Dispose()`** es el que se cerciora de que los recursos de la conexión se liberen adecuadamente, incluyendo el cierre de la conexión.

Con **`using`** nos aseguramos de que nuestra aplicación no va dejando por ahí recursos mal liberados y el código queda mucho más compacto y fácil de leer, además de que

no nos tenemos que preocupar de hacer la liberación de la manera adecuada ni de comprobar cosas como que una conexión está abierta antes de intentar cerrarla.

Por supuesto no tenemos por qué usarlo sólo para conexiones. Cualquier objeto cuyos recursos debamos liberar es candidato a **`using`**, por ejemplo objetos de GDI+ (como pinceles, brochas y demás..), recursos de transacciones (los objetos **`TransactionScope`**), etc...

Aprende a sacarle provecho ya que su funcionalidad es fundamental para crear aplicaciones escalables y que hagan un uso racional de los recursos del sistema.

# Cortocircuito de expresiones y su importancia

por **José Manuel Alarcón**

Se trata esta de una observación que, aunque obvia, en muchos casos puede pasar inadvertida y me parece por tanto interesante destacarla aquí.

**Las expresiones condicionales en C# se *cortocircuitan***. Esto quiere decir que se detienen las comprobaciones en cuanto no es necesario comprobar el resto de condiciones por haber hecho suficientes pruebas para determinar el resultado.

Me explico. Todo el mundo sabe que para que una condición **`AND`** proporcione un valor positivo (**`true`**) ambas condiciones que se comprueban deben ser positivas (**`true`**). Por ello, cuando C# encuentra una condición **`AND`**, y verifica que la primera de las dos condiciones que se comparan es falsa, automáticamente deja de comprobar la que falta, puesto que aunque fuese cierta el resultado será falso al serlo ya la primera y, entonces, ¿para qué seguir perdiendo el tiempo?

Lo mismo ocurre con los otros tipos de operadores booleanos. Por ejemplo, en una expresión **`OR`**, si la primera condición es cierta ya no se sigue comprobando el resto puesto que el resultado ya se sabe con la primera (verdadero **`OR`** lo-que-sea = verdadero).

Todo esto, aparte de las obvias implicaciones de rendimiento que proporciona, sirve también para salvar ciertas situaciones con expresiones elegantes.

Por ejemplo... Imaginemos que el resultado de llamar a una función, es una cadena (**`string`**). Si el proceso que genera dicha cadena fracasa o (por cualquier otro motivo) no puede ofrecer un resultado razonable, el método devuelve un nulo en lugar de una cadena. Nosotros queremos comprobar que el resultado de llamar al método no devuelva un nulo ni tampoco una cadena vacía, así que escribimos:

```
string s = MiMetodo();
if (s.Trim() == "" || s == null)
  {
        ....//Lo que sea
  }
```

Esta expresión es errónea y fallará cuando el método devuelva un nulo, aunque parecerá correcta en el resto de los casos (a priori los más comunes). ¿Por qué? Pues porque si '**s**' es un valor nulo, en cuanto se intente llamar a su método **`Trim()`** en la primera condición, se producirá un error puesto que no existe tal método para el objeto nulo.

La forma correcta de escribir esta comprobación es justo al revés:

```
string s = MiMetodo();
if (s == null || s.Trim() == "")
  {
        ....//Lo que sea
  }
```

De este modo si '**s**' es nulo la primera condición es cierta y por lo tanto no es necesario comprobar la segunda (hay un **`OR`**), y nunca se produce el error por intentar llamar a **`Trim().`** Si por el contrario la cadena es válida la primera condición es falsa y se debe comprobar la segunda, que es lo que necesitamos. La comprobación funciona perfectamente en todos los casos gracias al ***cortocircuito de expresiones lógicas***.

En lenguajes como Visual Basic clásico esta situación no se daba y había que hacer dos condicionales puesto que siempre se comprobaban todas las expresiones, sin cortocircuito. En Visual Basic .NET existe expresiones específicas que cortocircuitan los condicionales (**`AndAlso`**, **`OrElse`**), ya que los operadores comunes (**`And`**, **`Or`**, etc...) mantienen el comportamiento antiguo por compatibilidad.

Pero en C# le podemos sacar partido para ganar claridad y rendimiento.

Vale la pena tener en mente estas cuestiones en principio básicas pero muchas veces olvidadas.

# Métodos de Extensión en C#

por **Alberto Población**

Cuando deseamos añadir un nuevo método a una clase existente, típicamente editamos el código fuente de la misma, si disponemos de acceso al mismo, o creamos una clase derivada y añadimos en ella el nuevo método. Cuando esto no es factible por alguna razón (por ejemplo, si es una clase de tipo **`sealed`**), podemos recurrir a los métodos de extensión.

A la hora de definirlos, **los métodos de extensión tienen el mismo aspecto que los métodos estáticos**, con la diferencia de que su primer argumento lleva antepuesta la palabra clave **`this`** (que en este contexto no tiene nada que ver con el **`this`** que habitualmente utilizamos para tomar una referencia a la instancia actual).

A la hora de llamar al método, no escribimos ese argumento. En su lugar, el compilador toma la instancia del objeto sobre la que estamos llamando al método. Por ejemplo:

```
public static class Extensores
{
    public static int ContarPalabras(this string texto)
    {
        return texto.Split(' ').Length;
    }
}
```

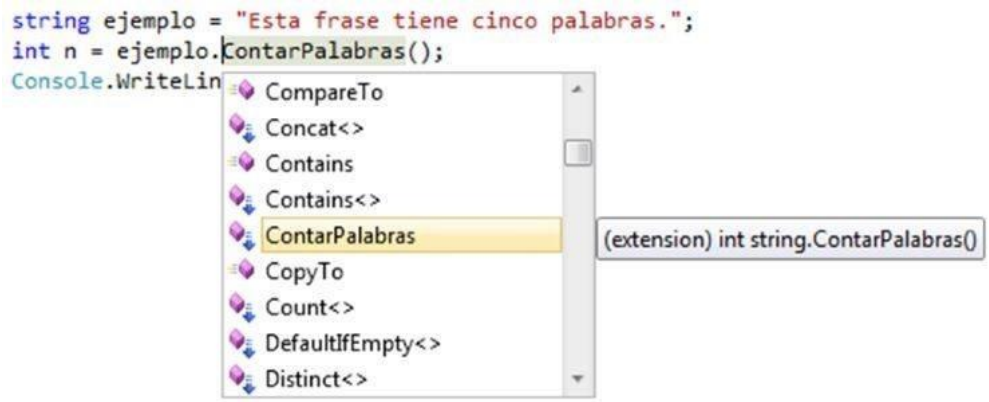
Podemos ver que se trata de un método estático definido dentro de una clase estática. **Para que funcione el método de extensión es obligatorio que tanto el método como la clase sean `static`.**

Para llamar al método, lo invocamos desde cualquier cadena, como si el método formase parte de la clase **`string`** (el tipo de argumento que va detrás del **`this`** en la declaración):

```
static void Main(string[] args)
{
    string ejemplo = "Esta frase tiene cinco palabras.";
    int n = ejemplo.ContarPalabras();
    Console.WriteLine("Hay {0} palabras.", n);
}
```

La llamada a **ContarPalabras** se realiza sobre la cadena ejemplo de la misma manera que si fuera alguno de los métodos estándar de la clase **string**. Desde el punto de vista del código llamante, no se nota ninguna diferencia respecto a los métodos nativos de la clase.

Al teclear el código en Visual Studio, *intellisense* muestra el método de extensión en la misma lista que el resto de los métodos, pero se ilustra con un icono ligeramente distinto para que podamos reconocer visualmente que se trata de un método de extensión.




En la anterior imagen vemos varios métodos de extensión (los que tienen una flecha azul), además del método **ContarPalabras** que nosotros hemos escrito. Estos métodos provienen de las librerías de **System.Linq**, que estaban referenciadas mediante un **using** al principio del programa del que se copió el ejemplo de la figura.

**No solo se pueden generar métodos de extensión sobre una clase, sino también sobre una interfaz**. De esta manera, todas las demás clases que hereden de esa clase, o que implementen esa interfaz, recibirán el método de extensión.

**Los métodos de extensión también pueden ser genéricos**. Por ejemplo, en **`System.Linq`** hay varios métodos de extensión que extienden la interfaz **`IEnumerable<T>`** de una forma similar a esta:

```
public static string Combinar<T>(this IEnumerable<T> ien)
{
    StringBuilder sb = new StringBuilder();
    foreach (T item in ien)
    {
        sb.Append(item.ToString());
    }
    return sb.ToString();
}
```

Después de definir este método, se puede invocar sobre cualquier objeto que implemente la interfaz **`IEnumerable<T>`**, por ejemplo, sobre un **`List<int>`**, que implementa **`IEnumerable<int>`**:

```
List<int> lista = new List<int>{ 9,8,7,6 };
Console.WriteLine(lista.Combinar());
```

Si se desea, se pueden añadir argumentos adicionales en los métodos de extensión. Después, en la llamada al método, esos argumentos se indican con normalidad entre los paréntesis, como si se hubiese declarado el método sin el argumento **`this`**.

```
public static int ContarPalabras(this string texto, char
separador)
{
    return texto.Split(separador).Length;
}
string ejemplo = "Hola, mundo. ¿Qué tal?";
int n = ejemplo.ContarPalabras(',');
Console.WriteLine(n); //Escribe "2"
```

Es sencillo añadir métodos de extensión en nuestros programas, pero **se recomienda no abusar de este mecanismo** ya que puede ocasionar una pérdida de claridad y mantenibilidad del código, al aparecer en las clases métodos que no figuran dentro del código fuente de las mismas.

Una aplicación común de los métodos de extensión consiste en extender interfaces tales como **`IEnumerable`** definiéndoles métodos tales como **`Where`**, **`Select`** u **`OrderBy`**, que luego permiten utilizar sentencias LINQ sobre cualquier objeto que implemente la interfaz en cuestión.

# Creación de métodos con un número arbitrario de argumentos en C#

por **José Manuel Alarcón**

¿Cómo construyo un método (función) que me permita pasarle un número cualquiera de parámetros?

Por ejemplo, imaginemos para simplificar que quiero construir una función que sume todos los enteros que se le pasen. Puedo definir varias versiones de la misma, cada una de las cuales tomando un número diferente de parámetros (un entero, dos enteros, tres, etc...). Hacerlo así, obviamente, aparte de ser una pérdida de tiempo es una cuestión bastante absurda: siempre encontraríamos un límite de parámetros, aparte de ser "*matar moscas a cañonazos*", como se suele decir.

La solución en C# es muy sencilla: basta con indicarle al compilador que nuestra función necesitará un número arbitrario de parámetros, que no sabemos de antemano cuántos serán. Esto se consigue haciendo uso de la palabra clave **params** en la definición del método. En nuestro ejemplo, una función **Suma** que tome un número cualquiera de enteros y los sume se definiría de la siguiente forma:

```
int Suma(params int[] sumandos)
 {
  int res = 0;
  for(int i = 0; i<sumandos.Length; i++)
  {
   res += sumandos[i];
  }
  return res;
 }
```

Fíjate que sólo tiene un parámetro de tipo matriz de enteros. Lo bueno es que al haber incluido la palabra clave **params** no es necesario construir una matriz de enteros y pasársela, sino que es el compilador quien lo hace por nosotros de manera

transparente simplificando mucho la llamada a la función, ya que basta con escribir algo así como:

```
int resultado = Suma(1,2,3,4,5,6,7);
```

o bien:

```
int resultado = Suma(1,2,3,4);
```

Como vemos se le puede pasar el número de argumentos que queramos y el compilador se encarga de convertirlo en una matriz y pasárselo a la función. Muy cómodo.

Fíjate que el método se comportará bien incluso si lo llamamos sin pasarle parámetro alguno:

```
int resultado = Suma();
```

devolvería un 0.

# ¿Y ahora qué?

¡Enhorabuena! Has llegado al final del eBook. Ahora ya tienes en tu poder 20 trucos y consejos por los que muchos veteranos en C# habrían pagado en sus inicios. Esperamos que los hayas encontrado de utilidad y los pongas en práctica.

## Los pasos para seguir aprendiendo C# y .NET

1- **Pásale este PDF a 3 amigos** (o, aún mejor, diles dónde descargarlo)
2- **Practica mucho.** No se aprende sólo leyendo o viendo videotutoriales.
3- **Si estás empezando con C# y .NET** o simplemente crees que necesitas una base más sólida para sacarle partido en tu trabajo, tenemos el curso de fundamentos de programación con C# y .NET que necesitas. Avanzarás más rápido y tendrás conocimientos más sólidos y estructurados. Además **te ahorrarás muchos meses de frustración** rebuscando en libros, blogs y tutoriales.
4- **Si ya programas habitualmente con C# y .NET**, **quizá te interese certificarte** para ser un profesional aún más valioso. En ese caso, prepara con nosotros el **examen de la Certificación Oficial de Microsoft 70-483**. Obtener esta certificación **puede marcar un punto de inflexión en tu carrera profesional** como programador.


Ver curso online de **Fundamentos en C# y .NET**


Ver curso online de **Certificación en C# y .NET**

## ¿Por qué aprender con nosotros?

Porque en campusMVP creamos cursos online de calidad contrastada cuyos autores y tutores son reconocidos expertos del sector. Ver todo el catálogo de cursos

**¿Quieres más razones?** Haz clic aquí

## Y no te olvides de nuestro blog...

En campusmvp.es/recursos encontrarás más eBooks gratuitos como este y otros interesantísimos recursos y noticias del mundo del desarrollo. ¡No dejes de visitarnos!