Por Guillermo 'Guille' Som
Visual Basic MVP desde 1997
www.elguille.info

Sobrecarga de operadores y conversiones personalizadas (con C#)

Una de las características que tiene el lenguaje C# desde su aparición es la posibilidad de crear sobrecargas de operadores, tanto aritméticos como lógicos, en nuestros propios tipos. En este artículo veremos algunos consejos que nos serán útiles a la hora de implementar esta característica.

» Sobrecarga de operadores

Todos usamos de forma habitual operadores en nuestros proyectos. Si queremos sumar dos enteros usamos el operador de la suma (**+**), que queremos comprobar si un valor es menor que otro, usamos el operador menor que (**<**). En principio podríamos pensar que esto es así porque sí, porque el compilador es inteligente y sabe cómo realizar esas operaciones. Bueno, sí, podría ser, pero realmente el compilador, o mejor dicho el motor en tiempo de ejecución, más conocido como CLR (*Common Language Runtime*), sabe cómo realizar esas operaciones porque se han definido, es decir, existe código que las realiza. Para todos, o casi todos, los tipos definidos en el .NET Framework existen funciones que le indica al compilador y al CLR cómo realizar dichas operaciones de suma, resta, etc.

Pero si queremos usar esos operadores en nuestros propios tipos (clases o estructuras), el compilador no sabrá cómo realizar dichas operaciones, por la sencilla razón de que no le hemos indicado cómo hacerlo.

Teniendo la definición de la estructura **Punto** del fuente 1, si hacemos una operación aparentemente sencilla como la suma de dos objetos de ese tipo, tal como se muestra en el siguiente trozo de código:

```
Punto p1 = new Punto(10, 25);
Punto p2 = new Punto(100, 22);
Punto p3 = p1 + p2;
```

El compilador se quejará y nos indicará que: "El operador '+' no se puede aplicar a operandos del tipo 'Punto' y 'Punto'". Y esto es así porque realmente el compilador no sabe qué hacer para sumar dos objetos del tipo **Punto**.

```
public struct Punto
{
      public int X;
      public int Y;
      //
      public Punto(int x, int y)
      {
            X = x;
            Y = y;
      }
      //
      public override string ToString()
      {
            return X.ToString() + ", " +
Y.ToString();
      }
}
```

Fuente 1. La estructura **Punto**

Si estás pensando que el compilador podría ser un poco más perspicaz y deducir cómo hacer esa operación; imaginemos que en lugar de sumar dos objetos del tipo **Punto**, quisiéramos sumar dos objetos de tipo **Cliente**... salvo que le enseñemos al compilador cómo manejar las probetas... tan solo la naturaleza será capaz de sumar dos *clientes* para obtener uno nuevo...

Por tanto, lo que tenemos que hacer es enseñar al compilador a sumar dos objetos del tipo que hemos definido, para que sea capaz de generar el código IL (*Intermediate Language*) correspondiente y de esta forma el CLR sepa también qué es lo que tiene que hacer

> **NOTA**
>
> A lo largo de este artículo usaremos una estructura (aunque también podría ser una clase) en la que implementaremos las sobrecargas y conversiones personalizadas; dicha estructura, llamada **Punto**, tendrá dos campos públicos **X** e **Y** los cuales representarán las coordenadas de un punto, la definición inicial es la mostrada en el fuente 1.

cuando se encuentre con esa operación de suma.

Cómo funciona la sobrecarga de operadores

Para entender mejor cómo funcionan las sobrecargas de los operadores, podemos comparar una sobrecarga con un método que realice la misma operación que el operador que queremos sobrecargar. Por ejemplo, si queremos crear un método que simule el operador de suma (+) podríamos escribir el método mostrado en el fuente 2.

```
public static Punto Add( Punto p1,
                         Punto p2)
{
return new Punto( p1.X + p2.X,
                  p1.Y + p2.Y );
}
```

Fuente 2. Un método para sumar dos objetos de tipo **Punto**

Como podemos comprobar, este método recibe por parámetro dos objetos del tipo **Punto** y devuelve un nuevo objeto de ese mismo tipo en el que hemos hecho la operación que estimemos oportuna para la suma de dos **Puntos**. También hemos declarado el método como estático, de forma que pertenezca al propio tipo y no a una instancia particular. Esto último no es un capricho, ya que es un requisito que los operadores sobrecargados sean métodos estáticos, por la sencilla razón de que los operadores se usarán de forma independiente, sin que haya ninguna instancia de por medio. Algo que es hasta lógico, porque dicha suma no se realizará sobre ninguna instancia en particular, sino que se hará sobre dos objetos cualesquiera del tipo en el que estamos definiendo el operador.

Al estar declarado el método **Add** como un método estático, para usarlo tendremos que indicar el nombre del tipo y el resultado asignarlo a una variable de ese mismo tipo, tal como se muestra en el fuente 3.

Si ejecutamos dicho código obtendremos la siguiente salida:

```
Punto p1 = 10, 25
Punto p2 = 100, 22
Punto p3 = 110, 37
```

Una vez que sabemos cómo sumar dos objetos del tipo **Punto**, vamos a definir el operador + (suma) para nuestro tipo, el cual quedaría tal y como se muestra en el fuente 4.

> **NOTA**
>
> Si quisiéramos usar el método **Add** como un método de instancia, (perteneciente al objeto creado en la memoria), lo recomendable es que sólo reciba un parámetro, el cual se usaría para incrementar el contenido del objeto actual, ese método de instancia tampoco debería devolver nada. La definición mostrada en el fuente 2 sería el código equivalente al operador sobrecargado para la suma, el cual nos servirá para comprender mejor cómo funcionan las sobrecargas de los operadores. De todas formas, en algunas ocasiones, sería recomendable que existiesen ciertos métodos estáticos, pertenecientes al propio tipo, que tuviesen su contrapartida con métodos de instancia.

```
Punto p1 = new Punto(10, 25);
Punto p2 = new Punto(100, 22);
Punto p3 = Punto.Add(p1, p2);
Console.WriteLine("Punto p1 = {0}", p1);
Console.WriteLine("Punto p2 = {0}", p2);
Console.WriteLine("Punto p3 = {0}", p3);
```

Fuente 3. Cómo usar el método **Add** de la estructura **Punto**

```
// operador +
public static Punto operator +( Punto p1,
                                 Punto p2)
{
   return new Punto( p1.X + p2.X,
                     p1.Y + p2.Y);
}
```

Fuente 4. La definición del operador suma

Si bien todo el código y la forma de declarar la sobrecarga de operadores y las conversiones personalizadas entre nuestros tipos y otros tipos (incluso del propio .NET Framework) están centradas en el lenguaje C#, todo lo aquí explicado también nos será de utilidad para aplicarlo a la nueva versión de Visual Basic 2005, la cual también nos permitirá sobrecargar operadores y realizar nuestras propias conversiones de tipos, pero eso será seguramente tema de otro artículo.

Como podemos comprobar, el código interno es el mismo que el del método **Add**, pero ahora al indicarle al compilador de C# qué es lo que tiene que hacer para sumar dos objetos de nuestro tipo, podemos usar el código del fuente 5, el cual, sin lugar a dudas, es mucho más intuitivo que el anterior, en el que usábamos el método **Add** y el resultado obtenido es el mismo que el del fuente 3.

```
Punto p1 = new Punto(10, 25);
Punto p2 = new Punto(100, 22);
Punto p3 = p1 + p2;
Console.WriteLine("Punto p1 = {0}", p1);
Console.WriteLine("Punto p2 = {0}", p2);
Console.WriteLine("Punto p3 = {0}", p3);
```

Fuente 5. Cómo usar el operador suma (+) de la estructura **Punto**

Tal como hemos podido comprobar es fácil definir nuestras propias sobrecargas de operadores, por tanto, ya estamos preparados para definir los operadores de nuestras clases y estructuras. La forma de declarar una sobrecarga de un operador siempre es la misma:

```
public static <valor devuelto>
 operator <operador>( <valor a la izquierda del operador>,
                      <valor a la derecha del operador>)
```

Esta sería la forma de definir los operadores binarios, es decir, los que operan con dos valores, uno de ellos, el que indicamos primero, es el que aparece en la parte izquierda del operado y el otro, el que indicamos segundo, es el que aparece en la parte derecha del operador, en cualquier caso, ambos parámetros deben ser por valor, es decir, no se permiten parámetros *out* o *ref*. Y, por supuesto, el valor devuelto por la función será el usado para indicar el resultado de dicha operación.

Crear sobrecargas en clases o en estructuras

En estos ejemplos estamos usando una estructura para definir los operadores sobrecargados. Pero realmente no hay ninguna razón especial para usar una estructura en lugar de una clase a la hora de decidir cual es mejor para crear nuestros tipos con sobrecargas de operadores. Lo que ocurre es que generalmente vamos a definir nuestros propios operadores en tipos que de alguna forma están pensados para manejar datos numéricos, y en estos casos es posible que el uso de una estructura sea más eficiente e intuitivo que el de una clase. Por supuesto, ya sabemos quién es el que finalmente decide que tipo de datos crear: nosotros mismos después de haber evaluado cual rinde mejor.

Algunos detalles a tener en cuenta

Antes de ver otras formas de definir los operadores de nuestros propios tipos, debemos conocer otros aspectos y efectos colaterales que se nos pueden presentar.

Por ejemplo, supongamos que queremos sumarle a un tipo **Punto** un valor entero, de forma que incrementemos el valor interno de **X**; podríamos definir el operador **+** (suma) de forma que reciba un parámetro del tipo **Punto** y un entero, tal como se muestra en el fuente 6.

```
public static Punto operator +(Punto p1, int x)
{
      return new Punto(p1.X + x, p1.Y);
}
```

Fuente 6. Definición de la suma para un tipo **Punto** y un entero

Esta sobrecarga permitiría escribir algo como:

```
Punto p1 = new Punto(100, 25);
Punto p2 = p1 + 15;
```

Pero si intentamos hacer esta otra operación:

```
Punto p2 = 15 + p1;
```

El compilador producirá un error, indicándonos que no hay ninguna sobrecarga que reciba esos parámetros. Y, en parte, es hasta lógico ya que el compilador no sabe cómo sumar un tipo entero y un tipo **Punto**, ¿verdad?

Podrías pensar que sí, que sí lo sabe, ya que lo hemos definido. Pero realmente, lo que hemos definido es cómo sumar un **Punto** y un entero, no al revés, y en este caso, el orden de los sumandos sí altera el valor de la suma.

Para comprender mejor porqué se produce este error, volvamos al código del método **Add**, el cual lo habíamos declarado como:

```
public static Punto Add(Punto p1, Punto p2)
```

Por tanto sabemos que sólo admitirá como parámetros dos valores de tipo **Punto**.

Ahora escribamos otra sobrecarga del método **Add** para que sea similar a la sobrecarga del operador + que acabamos de definir, el código sería el del fuente 7.

Viendo esta definición tenemos claro que la única forma de usarla sería pasándole un valor de tipo **Punto** y otro de tipo **int**, en este mismo orden, pero no al revés.

```
public static Punto Add(Punto p1, int x)
{
      return new Punto(p1.X + x, p1.Y);
}
```

Fuente 7. El método **Add** para sumar un **Punto** y un entero

Si quisiéramos pasar primero un valor de tipo entero y después el de tipo **Punto**, tendríamos que declarar el método **Add** tal y como se muestra en el fuente 8.

```
public static Punto Add(int x, Punto p1)
{
      return new Punto(p1.X + x, p1.Y);
}
```

Fuente 8. El método **Add** para sumar un entero y un **Punto**

Está más claro de esta forma, ¿verdad?, ya que ahora hemos definido una sobrecarga del método **Add** que acepte un entero como primer parámetro y un **Punto** como segundo. Por tanto ahora tendríamos tres sobrecargas del método **Add** que acepta los siguientes tipos de datos:

1- Dos valores de tipo **Punto**.
2- Un valor de tipo **Punto** y otro de tipo **int**.
3- Un valor de tipo **int** y otro de tipo **Punto**.

Pues lo mismo ocurre con las sobrecargas de operadores, el compilador sólo sabrá usar los que hayamos definido y cuando se encuentre uno para el que no tiene una definición, se quejará y nos mostrará un error. Por tanto, si queremos que nuestra estructura pueda sumar un valor de tipo **Punto** y un valor **int** sin importar el orden que se use, tendríamos que definir la sobrecarga correspondiente, tal como se muestra en el fuente 9.

```
public static Punto operator +(int x, Punto p1)
{
      return new Punto(p1.X + x, p1.Y);
}
```

Fuente 9. La sobrecarga para sumar un entero y un **Punto**

Algunas cosas que el compilador hace por nosotros

Pero, (sí, en las sobrecargas también hay peros), esta forma de actuar del compilador puede cambiar o puede funcionar con otros tipos para los que no hemos definido de forma explícita una sobrecarga.

NOTA

Cuando definamos operadores sobrecargados, debemos tener en cuenta que al menos uno de los parámetros debe ser del mismo tipo en el que lo estamos definiendo.

De igual forma, sólo podremos sobrecargar operadores existentes, es decir, si en C# no existe un operador para elevar un valor a una potencia, no podríamos "inventarlo". En la tabla 1, extraída de la documentación de Visual Studio .NET, se muestran tanto los operadores que podemos sobrecargar como los que no.

¿Cómo es posible?. Por la sencilla razón de que si el compilador sabe cómo hacer ciertas conversiones entre tipos, intentará hacerlas en el caso de que no se encuentre con una conversión específica. Por ejemplo, si queremos hacer la suma mostrada en el fuente 10, podríamos pensar que se producirá un error, ya que no hemos definido ninguna sobrecar-

Operadores	Posibilidad de sobrecarga
`+, -, !, ~, ++, --, true, false`	Estos operadores unarios sí se pueden sobrecargar.
`+, -, *, /, %, &, |, ^, <<, >>`	Estos operadores binarios sí se pueden sobrecargar.
`==, !=, <, >, <=, >=`	Los operadores de comparación sí se pueden sobrecargar, pero siempre se harán por parejas.
`&&, ||`	Los operadores lógicos no se pueden sob recargar, pero se evalúan con `&` y `|`, que pueden sobrecargarse.
`[]`	El operador de indización de matrices no se puede sobrecargar, pero se pueden definir indizadores.
`()`	El operador de conversión explícita de tipos no se puede sobrecargar, pero se pueden definir nuevos operadores de conversión.
`+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`	Los operadores de asignación no se pueden sobrecargar, pero `+=`, por ejemplo, se evalúa con `+`, el cual sí se puede sobrecargar.
`=, ., ?:, ->, new, is, sizeof, typeof`	Estos operadores no se pueden sobrecargar.

Tabla 1. Los operadores de C#

ga del operador suma que reciba como segundo operando un valor de tipo **short**.

```
short s = 15;
Punto p1 = new Punto(100, 25);
Punto p2 = p1 + s;
```

Fuente 10. Sumar un **Punto** con un valor **short**

El compilador buscará en la lista de sobrecargas del operador **+** uno que se adecue a esos tipos de datos, y realmente no encontrará una sobrecarga que coincida, pero sí que encontrará la del tipo **int**, y como resulta que el compilador sabe perfectamente cómo convertir de **short** a **int**, (esto se conoce como promoción de **short** a **int**), pues no habrá problemas, ya que hará la conversión de **short** a **int** y después usará el resultado de esa conversión con la sobrecarga que hemos definido en la que el segundo parámetro es un valor de tipo **int**.

Pero si en lugar de un **short** usamos un valor de tipo **long**, en este caso se producirá una excepción, ya que el compilador no puede convertir (promocionar) un valor **long** en uno de tipo **int** de forma implícita. Aunque siempre nos queda el recurso de hacer una conversión (**cast**) para convertir el valor **long** en un valor **int** y así poder hacer la suma, tal como podemos ver en el código mostrado en el fuente 11.

```
long n = 15;
Punto p1 = new Punto(100, 25);
Punto p2 = p1 + (int)n;
```

Fuente 11. Para sumar un **Punto** con un **long**, debemos convertirlo a entero

Este código es válido, al menos para el compilador, si bien en tiempo de ejecución si el contenido de la variable **n** es demasiado grande, se truncará a un valor entero. Por ejemplo, el código del fuente 12 asignará el valor **2112455033** a la propiedad **X** de la variable **p2**.

Conversiones personalizadas

Sabiendo que el compilador intentará hacer conversiones para usar las sobrecargas que tengamos definidas, nos encontramos con otra forma de crear sobrecargas automáticas de operadores: creando nuestras propias conversiones entre otros tipos y el que nosotros estamos creando.

```
long n = 12345678901234567890L;
Punto p1 = new Punto(100, 25);
Punto p2 = p1 + (int)n;
```

Fuente 12. C# sabe cómo manejar los *overflows* de números enteros

Tipos de conversiones

Como sabemos, existen dos tipos de conversiones: las *implícitas* y las *explícitas*, es decir, si no necesitan un *cast* o sí lo necesitan. Por ejemplo, el compilador de C# sabe convertir un **int** en **long** (promoción) y como sabe que esa conversión no produce pérdida de información, porque un entero de 64 bits (**long**) puede contener sin problemas un entero de 32 bits (**int**), por tanto lo hace de forma *implícita*. Sin embargo, lo contrario no es algo "natural", (convertir un **long** en un **int**), por la sencilla razón de que al ser la capacidad de un **long** muy superior a la de un **int**, es posible que algunos valores de un **long** no "quepan" en un **int**, porque, como acabamos de ver, un **int** se almacena en 32 bits de espacio, mientras que para almacenar un **long** necesitamos el doble: 64 bits; en estos casos, la conversión la haremos de forma *explícita*, es decir, tenemos que indicarle al compilador que sabemos que es posible que haya pérdida de información, pero aún así queremos hacer dicha conversión.

En nuestras clases (tipos) también podemos definir conversiones tanto implícitas como explícitas, por ejemplo, si queremos convertir un entero en un **Punto**, podríamos definir una conversión implícita de forma que el valor entero lo asignemos al campo **X** de nuestro tipo. La declaración sería la mostrada en el fuente 13.

> **NOTA**
>
> Las conversiones personalizadas se definen de la siguiente forma:
>
> ```
> public static <tipo de conversión>
>       operator <tipo devuelto>(<tipo a convertir>)
> ```
>
> Que podría servir para realizar una conversión (*cast*) de esta forma:
>
> ```
> (<tipo devuelto>)<tipo a convertir>
> ```
>
> Por tanto, siempre se ven involucrados dos tipos de datos, uno de los cuales debe ser del mismo tipo que define la conversión y el otro debe ser un tipo diferente, ya que no tendría mucho sentido convertir, por ejemplo, un tipo **Punto** en otro tipo **Punto**.

```
public static implicit operator
                        Punto(int x)
{
      return new Punto(x, 0);
}
```

Fuente 13. Conversión implícita de entero a **Punto**

Sin embargo la conversión de un objeto de tipo **Punto** a un entero sí que producirá pérdida de información, por tanto la conversión la deberíamos declarar como *explícita*, de forma que un usuario de nuestro tipo sea consciente de que posiblemente se pueda producir pérdida de información. La declaración de la conversión de **Punto** a **int** sería la mostrada en el fuente 14.

Esta sobrecarga la usaríamos de esta forma:

```
Punto p1 = new Punto(100, 25);
int x = (int)p1;
```

```
public static explicit operator
                        int(Punto p)
{
      return p.X;
}
```

Fuente 14. Conversión explícita de **Punto** a entero.

Con estas dos conversiones personalizadas, ya hemos "enseñado" al compilador de C# cómo convertir un **Punto** en un entero y viceversa, si bien al ser la última (la de entero a **Punto**) la que se usa de forma implícita, (sin conversión o *cast*), podemos aprovechar ese conocimiento del compilador para ahorrarnos algunas sobrecargas, por ejemplo las dos sobrecargas del operador **+** en las que se usa un valor **int** ya no serían necesarias, porque cuando el compilador se encuentre con un código como este:

```
Punto p2 = p1 + 10;
```

Buscará una sobrecarga del operador **+** en el que se utilice un **Punto** y un entero; al no encontrarla, comprobará si hay alguna otra forma de usar ese operador y se encontrará con la sobrecarga que suma dos valores de tipo **Punto**, y como ahora sabe cómo convertir de forma implícita un entero en un **Punto**, usará esa conversión personalizada para realizar la suma usando la sobrecarga que opera con dos valores de tipo **Punto**.

Estos pequeños detalles debemos tenerlos en cuenta cuando vayamos a crear sobrecargas de operadores y conversiones personalizadas en nuestros tipos.

Sobrecarga de operadores unarios

La sobrecarga del operador **+** que hemos visto, es la sobrecarga de un operador binario, es decir, un operador que utiliza (u opera con) dos valores, uno el que está a la izquierda del operador y el otro el que está a la derecha. Pero también podemos sobrecargar operadores unarios, como por ejemplo el operador de incremento (**++**) o de decremento (**--**); en estos casos sólo se usa un valor y éste debe ser del mismo tipo que el tipo que lo define, por tanto, el valor devuelto y el parámetro deben ser del mismo tipo en el que están definidos, en nuestro caso, del tipo **Punto**. En el fuente 15 podemos ver el código de la sobrecarga del operador **++** de nuestra estructura.

```
public static Punto operator ++(Punto p)
{
   return new Punto(p.X + 1, p.Y + 1);
}
```

Fuente 15. Sobrecarga del operador **++**

Esta sobrecarga nos servirá tanto cuando usemos el operador **++** como prefijo o sufijo: **p++** o **++p**, tal como podemos ver en el fuente 16.

```
Punto p1 = new Punto(100, 25);
Console.WriteLine("p1++ = {0}", p1++);
Console.WriteLine("++p1 = {0}", ++p1);
```

Fuente 16. Uso del operador de incremento

TRUCO

Sabiendo que el compilador siempre que sea necesario promocionará, (convertirá un tipo de menor capacidad en otro que tenga mayor capacidad de almacenamiento), podemos ahorrarnos algunas sobrecargas y conversiones personalizadas. Por ejemplo, si definimos una conversión de **long** a **Punto**, no será necesario crear las conversiones para **sbyte**, **byte**, **short**, **int** y **uint**; de igual forma, si definimos una conversión de un valor **double** a **Punto**, no será necesario crear una para **float**. Todo esto, siempre y cuando no necesitemos hacer comprobaciones extras en la forma en que se harán las conversiones.

> Los parámetros usados en las sobrecargas de operadores deben ser siempre por valor

Este código producirá el siguiente resultado:

```
p1++ = 100, 25
++p1 = 102, 27
```

La primera línea nos muestra el mismo valor que originalmente hemos asignado a la variable p1, pero como podemos comprobar en la segunda línea, dicho incremento se ha realizado. Para "comprender" el resultado, realmente hay que saber cómo funciona el operador ++, ya que cuando se usa como sufijo, se usa el valor que contiene y después se incrementa, sin embargo cuando se usa como prefijo (delante de la variable), primero se incrementa el valor y después se usa dicho valor.

¿Qué sentido tiene sobrecargar *true* o *false*?

Tal como pudimos comprobar en la tabla 1, tanto **true** como **false** se consideran operadores unarios, además de que se pueden sobrecargar. La primera impresión podría ser que podemos cambiar el valor que devuelven estos operadores, pero no es así, lo que sí podemos cambiar es el comportamiento de los mismos, es decir, cuando se considera que nuestro tipo devuelve un valor verdadero o uno falso. Por ejemplo, supongamos que nuestra estructura **Punto** la quisiéramos usar en una comparación o como resultado de una operación lógica, el compilador producirá un error si no definimos el comportamiento de estos operadores con respecto al contenido de nuestra clase o estructura.

Supongamos que en la estructura **Punto** queremos considerar que debe devolver un valor verdadero si cualquiera de los valores de los dos campos

NOTA

A pesar de que pueda parecer lógico (así era, si no recuerdo mal, en la beta de .NET Framework 1.0), que si hemos sobrecargado el operador **+** (suma), no sería necesario sobrecargar el operador de incremento (**++**), ya que la operación a realizar sería sumar uno al valor de la variable, pero el compilador de C# sólo usará las sobrecargas de **++** y **--** si lo hemos definido en nuestro código, de no ser así se producirá un error de compilación indicándonos que no existe una sobrecarga para el operador **++** o **--**.

X e **Y** contiene un valor distinto de cero; la declaración de la sobrecarga del operador **true** quedaría tal como se muestra en el fuente 17.

```
public static bool operator
                        true(Punto p1)
{
    return (p1.X != 0 || p1.Y != 0);
}
```

Fuente 17. Definición de la sobrecarga de **true**

Pero si sobrecargamos el operador **true**, también tendremos que sobrecargar el operador **false**, ya que estos, al igual que los operadores de comparación, deben sobrecargarse por pares, con idea de que el compilador sepa lo que debe hacer cuando se quiera comprobar si el contenido de un **Punto** es falso. Por tanto, la definición de la sobrecarga de **false** podría quedar tal y como se muestra en el fuente 18.

```
public static bool operator false(Punto p1)
{
      return (p1.X == 0 && p1.Y == 0);
}
```

Fuente 18. Definición de la sobrecarga de false

Sobrecarga de operadores de comparación

Otros operadores que podemos sobrecargar son los operadores usados para realizar comparaciones, aunque en estos casos las sobrecargas hay que hacerlas por pares, por tanto, si sobrecargamos el operador mayor que (**>**) también debemos sobrecargar el operador menor que (**<**).

El caso del operador de igualdad (**==**) sigue esta misma regla, y también debemos sobrecargar el operador de desigualdad (**!=**), pero como todos los tipos de .NET se derivan de **Object**, y ese tipo incluye un método relacionado con la igualdad de objetos: el método **Equals**, que sirve para comprobar si la instancia actual es igual al objeto pasado como parámetro, también deberíamos escribir nuestra versión de ese método. Aunque si no lo declaramos, no pasará nada, el código compilará, aunque recibiremos una advertencia indicándonos que deberíamos hacerlo:

```
Punto define el operador == o el operador !=
pero no reemplaza a Object.Equals(object o)
```

Pero si lo hacemos, las recomendaciones son de no llamar directamente al operador **==** desde el código de **Equals**, ya que el parámetro pasado a ese método puede ser un valor nulo y se produciría una excepción, aunque realmente no se llamaría a nuestra sobrecarga, por la sencilla razón de que al ser un valor nulo, la excepción se produciría antes. Lo importante es que deberíamos reemplazar ese método de forma que produzca el mismo resultado que si usáramos el operador de igualdad, si bien es cierto que en la mayoría de las ocasiones la implementación de **Equals** producirá el mismo resultado. Además, siguiendo con las recomendaciones de diseño, nunca debería producirse una excepción en este método ni en los operadores sobrecargados.

```
public override bool Equals(object obj)
{
      if(obj is Punto)
            return this == (Punto)obj;
      else
            return false;
}
```

Fuente 19. Definición del método **Equals**

En el código fuente 19 vemos cómo deberíamos sobrecargar el método **Equals**, es decir, comproba-

mos que el parámetro sea un valor de tipo **Punto** y lo comparamos usando nuestra sobrecarga del operador de igualdad, en caso de que no lo sea, devolvemos un valor falso.

Además, el método **GetHashCode** se utiliza para identificar de forma única a un objeto, y como el propio .NET lo utiliza para sus propias comprobaciones, también deberíamos crear nuestra propia versión de dicho método, sobre todo si nuestro tipo formará parte de colecciones tipo **Dictionary**, ya que .NET usará el valor devuelto por el método **GetHashCode** para crear valores únicos en las claves de los elementos de la colección. En el caso de que no sepamos cómo definir este método, la recomendación es usar el valor **GetHashCode** del método **ToString** que hayamos definido en nuestro tipo, el cual en la mayoría de los casos será más eficiente que el devuelto de forma predeterminada (**base.GetHashCode()**). Por ejemplo, si dos variables de tipo **Punto** contienen los mismos valores en los campos **X** e **Y**, el valor de **GetHashCode** será el mismo hayamos o no definido ese método, pero si los contenidos de esos campos son diferentes, al usar la implementación predeterminada, se devolverá siempre el mismo valor, sin embargo, si lo definimos tal y como se muestra en el fuente 20, el valor devuelto será distinto, tal como podemos comprobar si ejecutamos el código mostrado en el fuente 21.

```
public override int GetHashCode()
{
      return this.ToString().GetHashCode();
}
```

Fuente 20. Definición del método **GetHashCode**

```
Punto p1 = new Punto(100, 25);
Punto p2 = p1 + 2;
Console.WriteLine("p1.GetHashCode {0}",
                  p1.GetHashCode());
Console.WriteLine("p2.GetHashCode {0}",
                  p2.GetHashCode());
```

Fuente 21.Prueba para determinar el valor devuelto por **GetHashCode**

Operadores que no se sobrecargan, pero usan las sobrecargas definidas

Algunos operadores no se pueden sobrecargar, como por ejemplo el operador de asignación (**=**), ni otros compuestos como la asignación y la suma (**+=**), y similares, sin embargo, estos últimos no será necesario sobrecargarlos si ya tenemos la sobrecarga del operador que interviene junto con el de asignación; en el código de la clase **Punto**, hemos sobrecargado el operador para sumar, y como resulta que el compilador de C# siempre sabe cómo hacer una asignación, también sabrá cómo usar el operador **+=**, ya que esto es lo mismo que sumarle al objeto de la derecha lo que haya a la izquierda y después volver a almacenarlo en el que esté a la izquierda del operador, por tanto, si hacemos esto:

```
p1 += p2;
```

El compilador hará esta otra operación:

```
p1 = p1 + p2;
```

> **NOTA**
>
> Según las recomendaciones de diseño de .NET, el operador de igualdad (**==**) no debería reemplazarse en los tipos por referencia (clases). Por otra parte, si se sobrecargua algún operador aritmético es recomendable definir el operador de igualdad, sea un tipo por referencia o por valor.

> No todos los lenguajes de .NET soportan la sobrecarga de operadores, en esos lenguajes habrá que usar unos métodos equivalentes que el compilador crea de forma automática

Y como resulta que existe una sobrecarga para sumar dos objetos de tipo **Punto**, el compilador no tendrá problemas para efectuar esa operación, otra cosa bien distinta sería que no tuviésemos una sobrecarga adecuada, en cuyo caso nos mostrará el típico mensaje de error indicándonos que no existe una sobrecarga para la operación que queremos realizar.

Interoperabilidad con otros lenguajes de .NET

Hay que tener en cuenta que la sobrecarga de operadores sólo se podrán usar con lenguajes que la soporten, por ejemplo, si definimos una librería que contiene algún tipo en el que se han definido sobrecargas de operadores y conversiones personalizadas y dicha librería la queremos usar por ejemplo con

Visual Basic .NET, éste lenguaje no podrá aprovecharse de esas sobrecargas, por tanto deberíamos tener presente esta posibilidad y definir métodos equivalentes a las sobrecargas y conversiones que hemos definido. En la tabla 2, también extraída de la documentación de Visual Studio .NET, se muestran los operadores sobrecargables además de los métodos que deberíamos definir como alternativos y los métodos que crea el compilador para realizar esas operaciones, los cuales también podemos usar en esos lenguajes que no soporten la sobrecarga de operadores ni las conversiones personalizadas.

Por ejemplo, si creamos una librería de clases con la estructura **Punto** y la usamos desde un proyecto de Visual Basic .NET, para usar el operador suma tendríamos que usar un código parecido al mostrado en el fuente 22.

```
Dim p1 As New Punto(100, 25)
Dim p2 As Punto = Punto.op_Addition(p1, 10)
'
Console.WriteLine("p1= {0}, p2 = {1}", p1, p2)
```

Fuente 22. Código de VB usando el método equivalente a la sobrecarga de la suma

Operador	Método alternativo	Método creado por el compilador
No está definido	ToXxx o FromXxx	op_Implicit
No está definido	ToXxx o FromXxx	op_Explicit
+ (binario)	Add	op_Addition
- (binario)	Subtract	op_Subtraction
* (binario)	Multiply	op_Multiply
/	Divide	op_Division
%	Mod	op_Modulus
^	Xor	op_ExclusiveOr
& (binario)	BitwiseAnd	op_BitwiseAnd
|	BitwiseOr	op_BitwiseOr
<<	LeftShift	op_LeftShift
>>	RightShift	op_RightShift
==	Equals	op_Equality
>	Compare	op_GreaterThan
<	Compare	op_LessThan
!=	Compare	op_Inequality
>=	Compare	op_GreaterThanOrEqual
<=	Compare	op_LessThanOrEqual
--	Decrement	op_Decrement
++	Increment	op_Increment
- (unario)	Negate	op_UnaryNegation
+ (unario)	Plus	op_UnaryPlus
~	OnesComplement	op_OnesComplement

Tabla2. Los operadores, los métodos alternativos y los métodos internos

> **NOTA**
>
> La próxima versión de Visual Basic .NET soportará la sobrecarga de operadores y las conversiones personalizadas, por tanto con esa versión si se podrán utilizar las sobrecargas definidas en una librería creada con C#.

Pero como la estructura **Punto** también tiene un método **Add** para realizar las mismas operaciones, el código de Visual Basic quedaría mejor tal como vemos en el fuente 23.

```
Dim p1 As New Punto(100, 25)
Dim p2 As Punto = Punto.Add(p1, 10)
'
Console.WriteLine("p1= {0}, p2 = {1}", p1, p2)
```

Fuente 23. Código de VB usando el método **Add**

Por supuesto, también podremos usar el resto de operadores sobrecargados, pero siempre usando los métodos creados por el compilador, cuya nomenclatura vemos en la tercera columna de la tabla 2.

Conclusiones

La sobrecarga de operadores y las conversiones personalizadas nos permiten dar una nueva funcionalidad a los tipos creados por nosotros, pero como hemos comentado, algunos de los lenguajes de .NET no soportan esta característica, por tanto deberíamos proporcionar una funcionalidad paralela a la sobrecarga, al menos para evitar usar métodos poco amigables.

También debemos saber que a pesar de que en C# (y los lenguajes que soporten la sobrecarga de operadores y las conversiones personalizadas) podemos aprovechar las conversiones personalizadas para no tener que definir sobrecargas extras, en los lenguajes que no soporten las sobrecargas de operadores no podrán aprovecharse de ellas y para poder usarlas, tendremos que definirlas de forma explícita.

Espero que ahora tengas una idea más clara de cómo crear sobrecargas de operadores así como las conversiones personalizadas en C#, en otra ocasión veremos cómo utilizar estas características usando la nueva versión 2005 de Visual Basic.NET.

El código fuente con la definición de la estructura **Punto** así como los ejemplos de cómo usarla tanto desde C# como desde Visual Basic. lo puede bajar de *www.dotnetmania.com*.