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

Yo formateo, tú formateas... ¡que formatee él!

Cómo implementar clases que permitan formatos personalizados en .NET Framework

En este artículo veremos cómo implementar las diferentes versiones del método *ToString* para que, cuando sea preciso, el usuario de nuestras clases pueda utilizar los formatos personalizados que creamos conveniente aplicar a nuestros tipos de datos.

» Cualquier programador de .NET Framework habrá usado en más de una ocasión formatos que las clases *Console*, *String* o *StringBuilder* nos permiten; incluso con más frecuencia usará formatos que algunos tipos del propio .NET exponen mediante el método *ToString*. Esa misma funcionalidad "formateadora" podemos dársela a nuestras propias clases (o tipos) mediante la definición del método *ToString*, particularmente implementando el definido en la interfaz *IFormattable*.

Dar formato al estilo .NET

Cuando hablamos de dar formato, nos estamos refiriendo a la posibilidad de hacer que el contenido de un tipo de .NET Framework se muestre de una forma particular. Por ejemplo, si queremos mostrar o usar el contenido de un tipo de datos que contiene una fecha, es posible que nos interese que se muestre en formato fecha corta o fecha larga o, lo que será más habitual, que se muestre según nuestras preferencias. Por ejemplo, si la fecha actual es el 3 de enero de 2005 y queremos mostrarla usando solamente dígitos, seguramente la forma de hacerlo sería "03/01/2005", pero si queremos mostrarla de forma que el mes se muestre usando tres letras, la representación sería "03/ene/2005". Como vemos, todo dependerá de nuestras preferencias o de las preferencias del usuario, ya que para un norteamericano sería mejor que se la mostremos empezando por el mes seguido del día y por último el año: "1-3-2005". Todas estas formas diferentes de mostrar una misma fecha podemos hacerla gracias a que el tipo de datos que mantiene una fecha en .NET está preparado para que se puedan usar diferentes formatos. Normalmente esos formatos los indicaremos como una cadena que le pasaremos al método *ToString*, ya que este método es el encargado de devolver una representación visual del contenido de un tipo de datos. Para conseguir una cadena usando los tres casos de formato que hemos comentado tendríamos que usar el siguiente código:

```
DateTime.Now.ToString("dd/MM/yyyy")
DateTime.Now.ToString("dd/MMM/yyyy")
DateTime.Now.ToString("M-d-yyyy")
```

En estos casos lo que hacemos es pasar un formato al método *ToString*, ese método lo interpretará y devolverá la cadena correspondiente. Ni qué decir tiene que estos formatos están previamente establecidos y no son aleatorios, ni algo que existe porque sí, ya que, como sabemos, todo lo que ocurre en cualquier entorno de programación, ocurre porque alguien lo ha previsto y programado. De este último aspecto nos ocuparemos dentro de poco.

Los formatos de las clases de .NET

En los tipos de datos predefinidos en .NET podemos usar formatos que nos permiten dar distintos aspectos al contenido de los mismos. Anteriormente hemos visto tres ejemplos de formatos de fecha; en esos ejemplos hemos usado formatos mediante patrones definidos en la clase *DateTime*, (lo que la documentación de Visual Studio .NET llama "cadenas de formato personalizado *DateTime*"), pero también podemos usar cadenas con formatos predefinidos o formatos estándar; estos últimos suelen ser de una sola letra, por ejem-

plo, siguiendo con la fecha 3 de enero, si lo usamos de esta forma: `DateTime.Now.ToString("D")`, tendríamos este resultado: "lunes, 03 de enero de 2005".

Por supuesto existen otros formatos predefinidos para las fechas, los cuales nos permiten mostrar tanto fechas como horas de diferente forma.

Como es de esperar, también existen formatos para los tipos numéricos, y al igual que en las fechas nos encontramos con formatos ya predefinidos y otros que nos permiten componer un formato particular. Aunque debemos tener en cuenta que no todos los formatos son aplicables a todos los tipos numéricos. Por ejemplo, si trabajamos con valores de tipo entero, podremos usar el formato "x", (que sirve para devolver una representación en formato hexadecimal), pero ese formato no podremos aplicarlo a un valor de tipo *Decimal* o a uno de punto flotante.

En el código que acompaña a este artículo (en "Material de apoyo" en *www.dotnetmania.com*), podemos ver algunos ejemplos de los diferentes formatos que podemos usar tanto con las fechas como con los números.

¿Cómo definir una clase que sea "formateable"?

Los formatos que hemos comentado anteriormente, sólo son aplicables a los tipos numéricos y a los de fecha/hora. Cuando definimos nuestro propio tipo de datos, esos formatos no están disponibles. Al menos no lo estarán de forma predeterminada, por tanto, si queremos que nuestros propios tipos de datos acepten formatos mediante el método *ToString*, debemos programarlo nosotros.

Esa personalización de los formatos a aplicar a nuestros propios tipos de datos podemos hacerla de dos formas diferentes.

Definir el método *ToString* para dar formato

La primera forma de permitir que nuestros propios tipos se puedan usar para dar diferentes formatos al valor contenido, es creando nuestra propia sobrecarga del método *ToString*, de forma que acepte un parámetro en el que indicaremos el formato a aplicar. Así podremos usarlo de la misma forma que los tipos definidos en .NET.

Debido a que nuestros propios tipos siempre trabajarán con algún tipo de datos del propio .NET Framework, podremos usar el método *ToString* de ese tipo elemental para hacer el trabajo de dar formato al contenido interno de nuestro propio tipo, al menos salvo en el caso de que queramos dar formatos propios y únicos definidos en nuestro tipo. Pero para mantener las cosas simples (para que sea fácilmente comprensible), no vamos a inventarnos (al menos por ahora) ningún formato especial, sino que desde nuestro método llamaremos al del tipo usado internamente.

Para clarificar las cosas, vamos a definir una estructura a la que llamaremos `Moneda`, la cual tendrá una sola propiedad (realmente un campo público) llamada `Importe`, y que será del tipo *Decimal*. La declaración de esta estructura y el método *ToString* que nos permita darle formato será tal como se muestra en el fuente 1.

```
Public Structure Moneda
  Public Importe As Decimal
  '
  Public Overloads Function ToString(format As String) As String
      Return Importe.ToString(format)
  End Function
End Structure
```

Fuente 1. La estructura **Moneda**

Para usar los formatos proporcionados por esta estructura, podemos hacerlo de la misma forma que haríamos con un tipo de datos *Decimal*. De hecho sólo permitirá las mismas cadenas de formato que el tipo subyacente. En el fuente 2 tenemos un par de ejemplos de cómo usar los formatos de nuestra estructura.

```
Dim m1 As Moneda
m1.Importe = 1500.523D
'
Console.WriteLine(m1.ToString("f"))
Console.WriteLine(m1.ToString("#,###.##"))
```

Fuente 2. Uso de la estructura **Moneda** del fuente 1

Por supuesto, si pretendemos usar un formato no permitido por el tipo *Decimal*, obtendremos una excepción indicándonos que el formato no es válido, tal como ocurriría al intentar mostrarlo en formato hexadecimal (usando "x").

Definir nuestras propias cadenas de formato

Si por cualquier razón se nos ocurre que sería interesante crear un nuevo formato para nuestra estructura o dar una funcionalidad distinta a los ya existentes, e incluso redefinir o permitir usar algunos no válidos (como el que permite mostrarlo en hexadecimal), simplemente tendríamos que añadir el código correspondiente y asunto arreglado.

Para no alargar el artículo, y debido a que seguidamente vamos a definir nuestros propios formatos, en los proyectos `formatos3VB` y `formatos3CS` del código que acompaña al artículo puede ver un ejemplo tanto en Visual Basic como en C#.

Pero aunque hayamos definido nuestra propia sobrecarga del método *ToString* para que acepte una cadena en la que podamos indicar el formato, no siempre va a funcionar como esperamos. Aclaremos esto último: tal como hemos visto en el fuente 2, podemos usar *ToString* con el método *Write* (o *WriteLine*) de la clase *Console*, pero tanto esos métodos de la clase *Console* como el método *Format* de la clase *String* o el método *AppendFormat* de la clase *StringBuilder*, permiten que usemos una variable de nuestro tipo e incluso permiten que podamos indicar formatos especiales para aplicar a nuestros tipos, de tal forma que podamos componer una cadena en la que se incluirá el contenido de la variable usando diferentes representaciones. Sí, mejor verlo con un ejemplo. En el fuente 3 tenemos varias formas de usar los métodos que hemos comentado utilizando diferentes formatos.

```
Dim i As Integer = 1500
Console.WriteLine("{0}", i)
Console.WriteLine("{0:N}", i)
Console.WriteLine("{0,10}", i)

Dim sb As New System.Text.StringBuilder
sb.AppendFormat("El valor de i es '{0,12}'", i)
sb.AppendFormat("{0,11:N}", i)

Dim s As String = sb.ToString()
s &= String.Format("El valor de i es {0}", i)
Console.WriteLine(s)
```

Fuente 3. Ejemplos de formatos en algunos métodos de las clases de .NET

En el código del fuente 3 hemos usado una variable de tipo entero y como ocurre con el resto de los tipos incluidos en la librería de clases de .NET, el método *ToString* está implementado completamente, o mejor dicho, tiene todas las posibilidades de implementación del método *ToString*. Y en esta ocasión aún no estamos hablando de formatos especiales como ocurre cuando usamos, por ejemplo, `{0,11:N}`, sino de algo tan simple como el método *ToString* sin parámetros. Esta sería, o debería ser, una de las primeras cosas que tendríamos que hacer en nuestros propios tipos, ya que si no lo hacemos, nos podemos encontrar con algo inesperado al usar, por ejemplo, algo tan simple como una variable del tipo `Moneda` (definido en el fuente 1) con el método *WriteLine* de la clase *Console*: `Console.WriteLine(m1)`. Lo que este código mostrará no es el contenido (o valor) de la variable `m1`, sino el nombre completo de la estructura `Moneda`, (que incluye el espacio de nombres y el nombre del tipo).

¿Qué es lo que ocurre aquí? Lo que ocurre es que el runtime de .NET, cuando no sabe cómo obtener una representación de un tipo, llama al método *ToString* y si no está redefinido usará la implementación de la clase base de todos los tipos de .NET: *Object*, y esa implementación simplemente devuelve el nombre completo de la clase. Por tanto, lo primero que deberíamos hacer al crear nuestras clases es redefinir el método *ToString* para que devuelva algo "coherente" con el valor que almacena. En nuestro ejemplo del tipo `Moneda`, podríamos definirlo para que simplemente devuelva el valor de la propiedad `Importe`, tal como podemos ver en el fuente 4.

```
Public Overloads Overrides Function ToString() As String
    Return Importe.ToString()
End Function
```

Fuente 4. La declaración del método `ToString` del tipo `Moneda`

De esta forma nos aseguramos de que cuando alguien quiera obtener una cadena de una variable del tipo `Moneda`, realmente obtenga algo "válido"; en este caso el contenido de la propiedad `Importe`.

Ahora veamos que es lo que habría que hacer para poder dar soporte al resto de formatos que hemos usado en el fuente 3 e incluso cómo podemos definir los nuestros propios para que se puedan usar dentro de las llaves en las cadenas usadas en los métodos que permiten formatos especiales.

Definir formatos especiales para nuestros tipos

De igual forma que .NET utiliza el método *ToString* sin parámetros cuando no se indica qué método usar de una clase, al menos en los casos en los que se deben usar como cadenas, los métodos *Write/WriteLine* de la clase *Console* o los métodos *Format* y *AppendFormat* de las clases *String* y *StringBuilder* respectivamente cuando se usa un marcador de formato (el número entre las llaves) en el

que se indica un formato especial, como sería el caso al usar **{0, 11:N}**, en estos casos también se llama al método *ToString*, pero a una versión especial: el definido en la interfaz *IFormattable*.

La interfaz *IFormattable* sólo tiene un método: *ToString*, el cual recibe dos parámetros (o argumentos), el primero indica una cadena con el formato a aplicar y el segundo será un objeto con el proveedor del formato, el cual en la mayoría de los casos podremos obviar.

```
Public Structure Moneda
    Implements IFormattable
    '
    Public Importe As Decimal
    '
    Public Overloads Overrides Function ToString() As String
        Return Importe.ToString()
    End Function
    Public Overloads Function ToString(format As String) As String
        Return Importe.ToString(format)
    End Function
    Public Overloads Function ToString( _
        ByVal format As String, _
        ByVal formatProvider As System.IFormatProvider) As String _
            Implements System.IFormattable.ToString
        Return Me.ToString(format)
    End Function
End Structure
```

Fuente 5. El tipo **Moneda** con las tres versiones de **ToString**

La cadena que indica el formato será parecida a la usada en la versión simple, la que sólo recibe un argumento. Es más, podemos implementar este método *ToString* para que internamente llame al método *ToString* del tipo usado para almacenar el valor de nuestro tipo. En el fuente 5 podemos ver la definición simple de nuestro tipo en el que hemos implementado la interfaz *IFormattable*.

Debemos fijarnos que realmente no hemos hecho nada especial, simplemente indicarle al .NET que ahora nuestra clase implementa el método *ToString* definido en la interfaz *IFormattable* y que puede usarlo cada vez que un método necesite acceder a dicha implementación, pero esto es suficiente para que se puedan usar todos los formatos permitidos, en nuestro caso por el tipo *Decimal*, incluso el que indica la longitud o el número de posiciones que debe ocupar en la cadena, que es el valor numérico separado con una coma del marcador de posición, ya que ese comportamiento es algo "innato" de esos métodos formateadores, realmente el parámetro *format* del método *ToString* hace referencia al que se indica después de los dos puntos.

Tal como hemos definido la implementación de *ToString* de la interfaz *IFormattable*, usaríamos solamente los formatos permitidos por el tipo *Decimal*, pero si queremos definir otros formatos, tendremos que escribir el código que lo interprete y que devuelva lo que nos interese devolver. Esto es útil sobre todo para tipos de datos cuyo contenido interno no es tan obvio como el de un valor numérico, por ejemplo que tengamos una clase *Empleado* en la que nos interesaría usar formatos diferentes para mostrar los distintos valores que dicha clase pueda almacenar, pero para seguir con la simplicidad, veamos cómo definir algunos formatos especiales e incluso, como ya comentamos, redefinir algunos ya existentes o no válidos para el tipo usado internamente para almacenar el valor del importe de nuestra estructura **Moneda**. En el fuente 6 tenemos la definición de los dos métodos *ToString* que reciben algún parámetro.

Como recomendación final, decir que sólo deberíamos implementar los métodos "formateadores" en nuestras clases o tipos cuando realmente tenga algún sentido, ya que no siempre los usuarios de nuestros tipos los usarán para darles formatos especiales. El método que siempre deberíamos definir es el método *ToString* sin parámetros, ya que ese método tiene más usabilidad y en muchos casos nuestros usuarios querrán tener una representación imprimible del contenido de nuestro tipo.

```
Public Overloads Function ToString(ByVal format As String) As String
    Return Me.ToString(format, Nothing)
End Function
'
Public Overloads Function ToString( _
    ByVal format As String, _
    ByVal formatProvider As System.IFormatProvider) As String _
        Implements System.IFormattable.ToString
    If format Is Nothing OrElse format = "" Then
        Return Me.ToString()
    End If
    Select Case format.ToLower()
        Case "i"
            Return Decimal.Truncate(Importe).ToString()
        Case "if"
            Return Decimal.Truncate(Importe).ToString("#,###")
        Case "f", "m"
            Return Importe.ToString("#,###.####")
        Case "d"
            Return Importe.ToString("#.####")
        Case "x"
            Return CLng(Importe).ToString("x")
        Case Else
            Return Importe.ToString(format)
    End Select
End Function
```

Fuente 6. Los métodos **ToString** con parámetros del tipo **Moneda**