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

Yo clasifico, tú clasificas... ¡que clasifique él!

Cómo implementar clases que sean clasificables por .NET Framework

» Hay clases de .NET Framework, particularmente algunos tipos de colecciones, que nos permiten clasificar los elementos contenidos en ellas mediante el método **`Sort`**, pero debido a que el contenido de dichas colecciones pueden (y suelen) ser de tipos de datos dispares, entre ellos los creados por nosotros, tenemos que darle algún tipo de información para que pueda realizar una clasificación que esté acorde con el tipo de datos contenido en la colección; en este artículo veremos cómo podemos hacerlo.

Clasificar al estilo .NET

Los diseñadores de las clases de .NET Framework, entre otras cosas, sabían que los programadores de este entorno iban a necesitar funciones de clasificación de elementos; también sabían que posiblemente cada programador necesitara crear sus propias clases, las cuales no tenían porqué estar derivadas de otras ya prefabricadas que podrían estar preparadas "de fábrica" para que fuesen clasificables. Sabiendo todo esto lo que acordaron fue proporcionar de cierta característica a las clases para que se pudieran clasificar. De esta forma le daban al programador la libertad de poder definir sus clases como mejor les pareciera, pero para que todo funcionara bien, estos atrevidos programadores debían seguir unas pequeñas normas, así podrían ser libres de crear las clases a su antojo; eso sí, las clases que se fuesen a usar para ser clasificadas (y que, por tanto, formasen los elementos de una colección, o lista de objetos, que a priori serían de un tipo "indefinido", ya que sólo cada programador sabrá en cada ocasión y momento cúal es la clase que usará para que sea clasificada) deberían seguir unas sencillas normas de conducta. Esas normas las impone una interfaz; como sabemos, al usar una interfaz lo que hacemos es firmar una especie de contrato en el que nos comprometemos a usar de forma adecuada los miembros definidos en dicha interfaz. Una interfaz es una especie de clase abstracta que simplemente nos indica los métodos (o miembros) que nuestras clases deben implementar para que el contrato se formalice.

> Las clases que se vayan a usar para ser clasificadas deberán seguir unas sencillas normas de conducta. Esas normas las impone una interfaz.

Las clases/colecciones que implementan el método **`Sort`** (para clasificar), lo que hacen es usar el método **`CompareTo`** que debe estar implementado en las clases que se utilicen como elementos de una colección "clasificable". Para que dicho método **`CompareTo`** funcione como debe, ha de cumplir ciertas normas impuestas por la interfaz **`IComparable`**, cuyo único miembro es precisamente el método **`CompareTo`**, el cual recibe un parámetro de tipo **`Object`** (el tipo más genérico posible) y debe devolver un valor dependiendo de si ese objeto es menor, mayor o igual que el objeto con el que se compara, que no es ni más ni menos que el propio objeto que implementa dicho método.

¿Qué se consigue con todo esto? Que podamos tener la libertad de poder crear nuestros propios objetos "clasificables", de forma que, sabiendo qué es lo que contiene nuestro objeto, seamos libres de poder decidir cómo clasificarlos. Ya que no es lo mismo clasificar objetos de tipo **`String`**, **`Integer`**, etc., que clasificar, por ejemplo, objetos del tipo **`Cliente`** o **`CuentaBancaria`**. En los primeros no hay duda que se usará para clasificar el propio contenido de dichos objetos, pero en los tipos de datos que nosotros definamos, puede que no esté tan claro en qué debemos basarnos para realizar dicha clasificación. Por tanto, en esos casos seremos nosotros los que definamos qué es lo que tendremos en cuenta a la hora de clasificar o para ser más precisos: qué es lo que debemos tener en cuenta para, teniendo sólo dos elementos, decidir cual debe ir en primer lugar y cual debe ir en segundo lugar. Ya que esto es lo que realmente hace .NET, comparar dos objetos y así decidir cual de ellos debe ir antes.

« dotNetManía 22

```
Public Class Colega
 Private _nombre As String
 Private _apellidos As String
 ' Constructores
 Public Sub New()
 End Sub
 Public Sub New(_
       ByVal elNombre As String, _
       ByVal losApellidos As String)
   _nombre = elNombre
   _apellidos = losApellidos
 End Sub
 '
 Public Property Nombre() As String
   Get
     Return _nombre
   End Get
   Set(ByVal value As String)
      _nombre = value
   End Set
 End Property
 '
 Public Property Apellidos() _
                          As String
   Get
      Return _apellidos
   End Get
   Set(ByVal value As String)
      _apellidos = value
   End Set
 End Property
 '
 Public Overrides Function _
                ToString() As String
  Return _apellidos & ", " & _nombre
 End Function
End Class
```

Fuente 1. La clase **Colega**.

¿Cómo definir una clase que sea "clasificable"?

Para entender todo este galimatías, lo mejor es ver un caso práctico en el que tengamos una clase y queramos que se pueda usar, por ejemplo, como elemento de una colección **ArrayList** o cualquier otra que permita clasificar sus elementos.

Primero definiremos la clase; en este caso se llamará **Colega** y que, para simplificar, sólo tendrá dos propiedades: **Nombre** y **Apellidos**. También redefiniremos el método **ToString**, que es el método que algunas clases de .NET Framework utilizan cuando se quiere mostrar el contenido de un objeto, de forma que nuestra implementación del método **ToString** devuelva los apellidos y el nombre que contiene el objeto de la clase **Colega**.

Para facilitar la tarea de crear nuevos objetos, hemos definido un constructor al que se le puede pasar como parámetros el nombre y los apellidos del nuevo objeto que queremos crear (ver fuente 1).

Si utilizamos objetos de esta clase como elementos de una colección, por ejemplo, del tipo **ArrayList** y pretendemos clasificar su contenido, recibiremos una excepción indicándonos que no es posible clasificarlos, que .NET no sabe cómo clasificar objetos del tipo **Colega**.

Por ejemplo, podríamos tener código del fuente 2 para usar los objetos de la clase **Colega** y añadirlos a una colección del tipo **ArrayList**.

El error que se producirá no es demasiado aclaratorio aunque nos puede dar una pista: "Información adicional: Specified IComparer threw an exception."

Es decir, la interfaz **IComparer** ha lanzado una excepción. Esto nos puede hacer pensar que realmente la interfaz que nuestra clase debe implementar es **IComparer**, pero no, la interfaz que nuestra clase **Colega** debe implementar para que sea clasificable es **IComparable**, que es la que al fin y al cabo utiliza el método **Sort** para clasificar los elementos de la colección.

Por tanto, para que nuestra clase **Colega** pueda ser clasificada debemos implementar la interfaz **IComparable**. Si usamos una interfaz en nuestras clases, debemos implementar cada uno de los miembros de dicha interfaz, en el caso de VB, la implementación de los miembros de una interfaz llega aún más lejos, ya que de forma explícita hay que indicar que lo que queremos es precisamente definir un método que está ligado con una interfaz. Veamos el código que habría que añadir a la definición que hemos hecho de la clase **Colega** mostrada en el fuente 1.

Justo después de la definición de la clase añadiremos **Implements IComparable**. Con esto le estamos indicando al compilador que nuestra clase quiere firmar un contrato para poder usar los miembros definidos en la interfaz **IComparable**:

```
Public Class Colega
      Implements IComparable
```

A continuación debemos definir el método **CompareTo**, el cual recibe como parámetro un objeto de tipo **Object** que será el que debemos usar para comparar

```
'una colección del tipo ArrayList
Dim al As New ArrayList

'Agregamos algunos datos
al.Add(New _
     Colega("Pepe","Ruiz"))
al.Add(New _
     Colega("Pepe","Lopez"))
al.Add(New _
     Colega("Pepe","Sancho"))
al.Add(New _
     Colega("Maria","Ruiz"))
al.Add(New _
     Colega("Maria","Castillo"))
al.Add(New _
     Colega("Maria","Rodriguez"))

'los clasificamos
al.Sort()

'los mostramos
For Each c As Colega In al
    Console.WriteLine(c)
Next
```

Fuente 2. El código de ejemplo para usar y clasificar los objetos del tipo **Colega**

con nuestro objeto, es decir debemos comparar dicho parámetro con la instancia que se ha creado de nuestra clase **Colega**.

Empecemos viendo la definición de dicho método:

```
Public Function CompareTo( _
       ByVal obj As Object)_
       As Integer Implements_
       System.IComparable.CompareTo
```

Como podemos comprobar, es una función que devuelve un valor de tipo **Integer**, que recibe como parámetro un objeto del tipo **Object** y que, de forma explícita, indica que está implementando el método **CompareTo** de la interfaz **IComparable**.

El problema con el que nos encontramos es que el parámetro es de tipo **Object**, no del tipo **Colega**, (de este detalle nos ocuparemos en el siguiente párrafo), para devolver el valor de la comparación nos apoyaremos en otra función, la cual devolverá cero si los dos objetos son iguales, menor que cero si la instancia actual es menor que la que se indica en el parámetro y mayor que cero si la instancia actual es mayor que la indicada en el parámetro.

Para poder comparar la instancia actual del tipo **Colega** con la indicada en

el parámetro del método **CompareTo**, debemos suponer (o dar por sentado) que dicho parámetro realmente es otro objeto del tipo **Colega**, pero como .NET usa el tipo **Object** de forma genérica para indicar cualquier tipo de objeto, (con idea de que el mismo método sirva para cualquier tipo de clase, cosa que cambia en la versión 2.0 de .NET Framework y el uso de *generics* y que podrás ver en el próximo número de **dotNetManía**), por tanto lo que debemos hacer es una conversión del tipo **Object** a nuestro propio tipo, en este caso **Colega**. Para ello usaremos la instrucción **CType** o **DirectCast** de forma que podamos tener una clase adecuada al tipo que queremos comparar:

```
Dim c1 As Colega = CType(obj, Colega)
```

Ahora la variable **c1** será la "versión" de tipo **Colega** del parámetro que ha recibido el método **CompareTo**. Una vez que ya tenemos 2 objetos del mismo tipo (uno el que representa a la instancia que implementa el método **CompareTo** y otro el que se ha pasado como parámetro a esta función) podemos comprobar cual es mayor.

En lugar de hacer tres comparaciones para saber si son iguales, mayor o menor, vamos a usar el método estático (o compartido) **Compare** de la clase **String** que se encargará de devolver el valor adecuado:

```
Return String.Compare( _
       Me.ToString, c1.ToString)
```

En este caso hemos usado el valor devuelto por el método **ToString**, ya que dicho método lo que hace es devolver el apellido seguido del nombre, por tanto será totalmente válido para realizar la comparación que indicará si un objeto es mayor o no que otro.

Por supuesto en el valor devuelto puedes usar la comparación que creas oportuno, si sólo quieres clasificar teniendo en cuenta los apellidos podrías hacer esto:

```
Return String.Compare(_apellidos, _
                      c1.Apellidos)
```

En el caso de que quieras que se clasifiquen de forma descendente, puedes invertir el orden de los valores que le pasamos a **String.Compare**:

```
Return String.Compare(c1.Apellidos, _
                      _apellidos)
```

¿Qué ocurre si quiero clasificar elementos que no han sido preparados para ser clasificados?

Pero se nos puede dar el caso en el que queremos usar una clase que no haya sido preparada como clasificable pero que nos interese que sí lo sea y, para complicar la situación, de la que no podamos derivar nuevas clases ni de modificar el comportamiento interno de dicha clase. En estos casos, podemos echar mano de una de las sobrecargas del método **Sort**, en la que se puede indicar una clase que implemente la interfaz **IComparer** y que reciba como parámetros dos objetos del mismo tipo del que queremos clasificar. En este caso podríamos crear una nueva clase que implemente dicha interfaz y el único método que tendrá la clase será el método **Compare**, el cual recibirá dos objetos por parámetro del tipo contenido en la colección y que es el que queremos clasificar.

```
Public Class ColegaComparer
 Implements IComparer
  '
 Public Function Compare(_
              ByVal x As Object, _
              ByVal y As Object)_
       As Integer Implements _
System.Collections.IComparer.Compare
  Dim c1 As Colega = _
                  CType(x, Colega)
  Dim c2 As Colega = _
                  CType(y, Colega)
  '
  Return String.Compare( _
         c1.ToString, c2.ToString)
 End Function
End Class
```

Fuente 3. La clase **ColegaComparer** para usarla para clasificar elementos tipo **Colega**

Suponiendo que tenemos la definición de la clase **Colega** del fuente 1, (a la que aún no habíamos añadido la implementación de la interfaz **IComparable**), y son objetos de esa clase los que queremos usar para añadir a la colección del tipo **ArrayList**, pero queremos que puedan ser clasificados, tendremos que crear una clase que implemente la interfaz **IComparer** y cuyo método **Compare** haga una comparación entre dos elementos del tipo **Colega**. (Ver el fuente 3 para mayor claridad).

Como vemos, esta clase sólo tiene un método que es el que se define en la interfaz **IComparer**. Para usar esta clase junto con el método **Sort** de la colección **ArrayList** tendremos que hacer:

```
Dim miComparer As New ColegaComparer
al.Sort(miComparer)
```

Es decir, pasamos como parámetro un objeto creado de la clase que sabe cómo clasificar dos objetos del tipo que estamos usando como elemento de la colección. Fíjate que realmente el objeto pasado como parámetro al método **Sort**, no tiene ninguna relación directa con ninguno de los objetos que contiene la colección; simplemente .NET Framework lo usará pasándole como parámetros al método **Compare** dos objetos del tipo **Colega** y dicho método se encargará de devolver el valor adecuado.

Comprobar que el tipo a clasificar es del tipo adecuado

En estos ejemplos no se ha usado ningún tipo de validación de datos correctos, es decir, hemos dado por supuesto que todo el contenido de la colección son objetos del tipo **Colega**, pero si por alguna casualidad, de esas que nunca se suelen producir, la colección contuviera elementos de distintos tipos, el código del fuente 2 así como el modificado para usar la interfaz **IComparer**, fallaría al intentar convertir un objeto que no es del tipo **Colega** al tipo **Colega**.

En estos casos, lo mejor es utilizar un **Try/Catch** para curarnos en salud. Que se produce un error al convertir el parámetro, pues en lugar de dejar que se quede "parada" la aplicación, podemos devolver un valor cero, indicando que "no sabemos" cúal es menor, por tanto asumimos que son iguales.

Por supuesto que aquí también podríamos tener en cuenta otros tipos de objetos, por ejemplo, que alguno de ellos sean del tipo **String** o de algún otro tipo "conocido" por nosotros; en esos casos podríamos hacer una comprobación del tipo de datos que recibimos como parámetro antes de realizar la conversión (o *cast*).