Capítulo 4
Características generales del lenguaje (III)

Introducción

En este capítulo continuamos con las novedades de Visual Basic 9.0, pero en esta ocasión solo las podremos usar con la versión 3.5 .NET Framework.

Estas características (al igual que casi todas las novedades de Visual Basic 9.0) tienen su razón de ser en todo lo referente a LINQ; pero antes de ver qué es LINQ y cómo usarlo desde nuestro lenguaje favorito, veámoslas con detalle para que nos resulte más fácil comprender lo que he dejado para el final de este repaso en profundidad a las novedades de Visual Basic 9.0.

Las novedades que veremos en este cuarto capítulo son:

- Los tipos anónimos.
- Las expresiones *lambda* o funciones anónimas.

Tipos anónimos

Esta característica nos permite crear nuevos tipos de datos (clases) sin necesidad de definirlos de forma separada, es decir, podemos crear tipos de datos "al vuelo" y usarlos, en lugar de crear una instancia de un tipo existente.

Esta peculiaridad nos va a permitir crear clases (los tipos anónimos deben ser tipos por referencia) cuando necesitemos crear un tipo de datos que tenga ciertas propiedades y que no necesiten estar relacionadas con una clase existente. En realidad, esto es particularmente útil para la creación de tipos de datos "personalizados" devueltos por una consulta de LINQ, aunque no necesitamos saber nada de LINQ para trabajar con estas clases especiales, por eso en este capítulo veremos cuáles son las peculiaridades de los tipos anónimos, pero usados de una forma más generalizada; cuando lleguemos al capítulo de LINQ, retomaremos este tema para entenderlos mejor en el contexto de LINQ.

Como sabemos, todos los tipos de datos pueden definir métodos, propiedades, eventos, etc. Sin embargo, en los tipos anónimos solo podemos definir propiedades. Las propiedades de los tipos anónimos pueden ser de solo lectura o de lectura/escritura. Este último aspecto, diferencia los tipos anónimos de Visual Basic de los que podemos crear con C#, ya que en C# las propiedades siempre son de solo lectura.

Los tipos anónimos, al igual que el resto de clases de .NET, se derivan directamente de **Object**, por tanto, siempre tendrán los métodos que esa clase define, y ésos serán los únicos métodos que tendrán. El compilador creará definiciones de esos métodos, que tendrán en cuenta las propiedades del tipo anónimo, pero como veremos en un momento, esas definiciones dependerán de cómo declaremos las propiedades.

Definir un tipo anónimo

Como su nombre indica, los tipos anónimos no están relacionados con ningún tipo de datos existente; éstos siempre se crean asignando el tipo a una variable y ésta infiere el tipo, que en realidad no es anónimo, ya que el compilador de Visual Basic le da un nombre; lo que ocurre es que no tenemos forma de saber cuál es ese nombre, por tanto, nunca podremos crear nuevas variables de un tipo anónimo de la forma tradicional, puesto que al ser anónimo, no podemos usar la cláusula **As** para definirlos. Para clarificar la forma de crear los tipos anónimos veamos un ejemplo.

En el listado 4.1 tenemos la definición de un tipo anónimo, el cual asignamos a una variable. Como vemos en ese código, para crear el tipo anónimo tenemos que usar la inicialización de objetos, pero a diferencia de usarla como vimos en el capítulo anterior, no podemos indicar el tipo de datos, ya que es un tipo anónimo.

```
Dim ta1 = New With {.Nombre = "Guille", .Edad = 50}
```

Listado 4.1. Definición de un tipo anónimo

El código del listado 4.1 crea una clase anónima con dos propiedades; los tipos de datos de dichas propiedades se infieren según el valor que le asignemos; esta inferencia de tipos siempre se hará de la forma adecuada, incluso si tenemos desactivado **Option Infer**.

La variable ***ta1*** es una instancia del tipo anónimo que acabamos de definir y tendrá dos propiedades, una llamada ***Nombre*** que es de tipo **String** y la otra, ***Edad*** que es de tipo **Integer**.

Cuando definimos un tipo anónimo de esta forma, las propiedades son de lectura/escritura, por tanto, podemos cambiar el contenido de las mismas, tal como vemos en el listado 4.2.

```
ta1.Edad = 51
```

Listado 4.2. De forma predeterminada, las propiedades de los tipos anónimos son de lectura/escritura

Como ya comenté, Visual Basic redefine los métodos heredados de **Object** para adecuarlos al tipo de datos que acabamos de definir. De estos métodos, solo la implementación de **ToString** siempre tiene el mismo formato, que consiste en devolver una cadena cuyo contenido tendrá esta forma: **{ Propiedad = Valor[, Propiedad = Valor] }**, es decir, dentro de un par de llaves incluirá el nombre de la propiedad, un signo igual y el valor; si hay más de una propiedad, las restantes las mostrará

separadas por una coma. Después de la asignación del listado 4.2, una llamada al método **ToString** de la variable ***ta1*** devolverá el siguiente valor:

```
{ Nombre = Guille, Edad = 51 }
```

¿Cómo de anónimos son los tipos anónimos?

En realidad, los tipos anónimos no son "totalmente" anónimos, al menos en el sentido de que sí tienen un nombre, lo que ocurre es que ese nombre no estará a nuestra disposición, ya que el compilador genera un nombre para uso interno. El tipo anónimo definido en el listado 4.1 tendrá el nombre ***VB$AnonymousType_0***, pero ese nombre solo es para uso interno, por tanto, no podremos usarlo para crear nuevas clases de ese mismo tipo (incluso si pudiéramos definir variables con ese nombre, el compilador no lo permitiría, ya que se usan caracteres no válidos para los nombres de variables o tipos, recordemos que el signo $ se utiliza en Visual Basic para indicar que una variable es de tipo **String**, por tanto, al encontrar ese signo acabaría la declaración del nombre del tipo).

Pero si nuestra intención es crear más variables del mismo tipo anónimo, lo único que tendremos que hacer es definir otra variable usando un tipo anónimo con las mismas propiedades, de esa forma, el compilador reutilizará el tipo anónimo que previamente había definido.

En el código del listado 4.3 la variable ***ta2*** también tendrá una instancia de un tipo anónimo, pero debido a que las propiedades se llaman igual, son del mismo tipo y están en el mismo orden que el definido en el listado 4.1, el compilador no creará otro tipo anónimo, sino que reutilizará el que definió para la variable ***ta1***.

```
Dim ta2 = New With {.Nombre = "David", .Edad = 28}
```

Listado 4.3. Este tipo anónimo es el mismo que el del listado 4.1

Cada vez que el compilador se encuentra con un tipo anónimo que tiene las mismas propiedades (nombre, tipo y orden de declaración) de otro definido previamente en el mismo ensamblado, usará el que ya tiene definido, pero si cualquiera de esas tres condiciones cambia, creará un nuevo tipo.

Por ejemplo, el compilador creará un nuevo tipo para el que definimos en el listado 4.4, porque aunque tiene dos propiedades que se llaman igual y son del mismo tipo que el de los listados anteriores, el orden en que están definidas esas propiedades es diferente, por tanto, lo considera como otro tipo de datos.

```
Dim ta3 = New With {.Edad = 24, .Nombre = "Guille"}
```

Listado 4.4. Si el orden de las propiedades es diferente, se crea otro tipo anónimo

La explicación a esta extraña forma de actuar es porque, en realidad, el compilador define los tipos anónimos como tipos *generic* y en estos dos casos concretos, la definición de los dos tipos es como vemos en el listado 4.5. Ese código es el generado por el compilador, que como sabemos lo genera usando el lenguaje intermedio de Microsoft (MSIL, *Microsoft Intermediate Language* o IL) que es el que finalmente compila el CLR cuando ejecutamos la aplicación, y que tiene una sintaxis muy parecida a C#; en el listado 4.6 tenemos el equivalente al estilo de cómo lo definiríamos en Visual Basic.

```
class VB$AnonymousType_0`2<string, int32>
class VB$AnonymousType_1`2<int32, string>
```

Listado 4.5. Definición de los dos tipos anónimos en lenguaje IL

```
Class VB$AnonymousType_0(Of String, Integer)
Class VB$AnonymousType_1(Of Integer, String)
```

Listado 4.6. Definición de los dos tipos anónimos al estilo de Visual Basic

En realidad, la definición de los tipos anónimos no usan los tipos de datos explícitos de las propiedades, ya que el compilador define esos tipos como *generic*. Lo que en realidad ocurre, es que, aunque las propiedades se llamen igual, al estar definidas en un orden diferente, la asignación de los tipos *generic* usados se invertirían y el resultado no sería el deseado.

Para que lo entendamos mejor, el seudocódigo al estilo de Visual Basic de la definición de esos dos tipos sería como el mostrado en el listado 4.7, y como vemos, la propiedad ***Edad*** de la primera clase es del tipo ***T0***, mientras que en la segunda clase es del tipo ***T1***, por tanto, si se usara un solo tipo anónimo, los tipos usados no coincidirían en las dos declaraciones.

```
Class VBAnonymousType_0(Of T0, T1)
    Public Nombre As T0
    Public Edad As T1
End Class

Class VBAnonymousType_1(Of T0, T1)
    Public Edad As T0
    Public Nombre As T1
End Class
```

Listado 4.7. Los tipos anónimos se definen como tipos generic y cada propiedad es del tipo indicado según el orden de las declaraciones

Si solo tuviéramos la primera clase (***VBAnonymousType_0***) y definimos dos variables, pero en la segunda declaración invertimos los tipos de los parámetros, el compilador nos daría un error, tal como vemos en la figura 4.1, ya que los valores asignados a las propiedades no son de los tipos correctos.

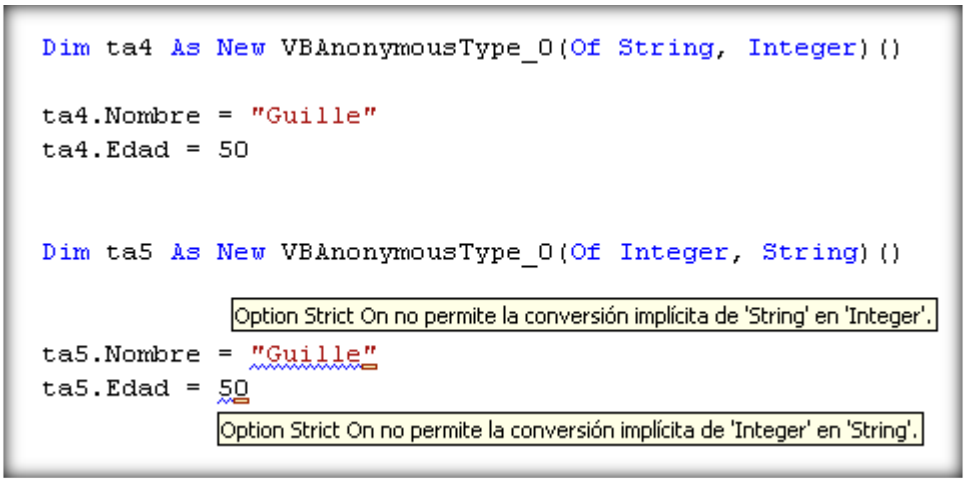

Figura 4.1. Error al asignar los valores a las propiedades del supuesto tipo anónimo

Como truco nemónico, yo pienso en los tipos anónimos como si fueran sobrecargas de un mismo método, si se puede crear una nueva sobrecarga, es que el tipo anónimo será diferente; y si puedo usar una de las sobrecargas existentes es que ya existe un tipo anónimo que se puede reutilizar (ya sabemos que en las sobrecargas lo que se tiene en cuenta son el número de parámetros y los tipos de esos parámetros, y solo se permiten sobrecargas cuando la firma no coincide con una existente), pero como veremos en la siguiente sección, al evaluar esas "sobrecargas" debemos tener en cuenta otras consideraciones, entre las que se incluye que el nombre de los parámetros también sea el mismo (y esté definido en el mismo orden).

Definir un tipo anónimo con propiedades de solo lectura

De los métodos definidos en la clase **Object**, que son los únicos que tendrán un tipo anónimo, hay dos que se redefinen de una forma especial cuando el tipo anónimo define propiedades de solo lectura: **Equals** y **GetHashCode**.

El primero servirá para comparar dos instancias de tipos anónimos, ya que la comparación de esas instancias se realizará comprobando solo el contenido de las propiedades de solo lectura (el resto de propiedades se descartará en la comparación). Para realizar esa comparación se utiliza el valor generado por el método **GetHashCode** en cada una de las propiedades de solo lectura.

Para definir propiedades de solo lectura, tenemos que anteponer el modificador **Key** a la definición de la propiedad. Podemos definir tantas propiedades de solo lectura como queramos.

En el listado 4.8 tenemos la definición de un tipo anónimo con una propiedad de solo lectura.

```
Dim ta6 = New With {Key .ID = "009", .Nombre = "VB9"}
```

Listado 4.8. La propiedad ID es de solo lectura

Como es lógico, el contenido de las propiedades de solo lectura no lo podemos modificar. El valor se asigna en el constructor de la clase y permanece inalterado durante toda la vida de esa instancia. El constructor siempre lo define el compilador, y en todos los casos, este constructor recibe tantos parámetros como propiedades hayamos definido en el tipo anónimo. Pero el constructor es solo para uso interno del compilador, ya que nosotros no tenemos acceso a la creación de nuevas instancias de un tipo anónimo, (salvo cuando los definimos), esto no cambia ni aun cuando definamos propiedades de solo lectura.

Cuando definimos propiedades de solo lectura, esa característica también la tendrá en cuenta el compilador para decidir si el tipo se puede reutilizar o se debe crear uno nuevo. Por tanto, debemos añadir un nuevo condicionante a la lista que vimos en la sección anterior para saber cuándo se crea un nuevo tipo anónimo o se utiliza uno que hayamos definido anteriormente.

Si definimos un tipo anónimo como el mostrado en el listado 4.9, el compilador creará un nuevo tipo y no reutilizará el que definimos para asignar la variable ***ta6***, ya que, si bien tiene las mismas propiedades y están en el mismo orden, la primera no es de solo lectura, por tanto, no coinciden las definiciones.

```
Dim ta7 = New With {.ID = "009", .Nombre = "VB9"}
```

Listado 4.9. Este tipo anónimo es diferente al definido en el listado 4.8

Como ya comenté, cuando comparamos dos instancias de tipos anónimos, dicha comparación se hará sobre las propiedades de solo lectura (las definidas con **Key**), por tanto, la comparación realizada en el código del listado 4.10 devolverá un valor falso, sin embargo, la comparación del listado 4.11 devolverá **True** (ejercicio: argumentar por qué esas dos comparaciones devuelven esos valores).

```
Dim sonIguales = ta6.Equals(ta7)
```

Listado 4.10. Esta comparación de igualdad fallará

```
Dim ta8 = New With {Key .ID = "009", .Nombre = "Visual Basic 9.0"}
sonIguales = ta6.Equals(ta8)
```

Listado 4.11. Esta comparación devolverá un valor verdadero

> ***Nota***
>
> *Cuando queramos comparar dos instancias de tipos anónimos tendremos que usar el método Equals, ya que el operador de igualdad (=) no está sobrecargado para comparar tipos anónimos.*

La explicación de por qué las instancias de las variables ***ta6*** y ***ta7*** no son iguales es porque ambos tipos anónimos son diferentes, ya que en el primero definimos la propiedad ***ID*** de solo lectura, mientras que en el segundo es de lectura/escritura.

Sin embargo, las instancias asignadas a las variables ***ta6*** y ***ta8*** son del mismo tipo, y el valor de la propiedad de solo lectura es el mismo en ambas instancias, y como el contenido de las propiedades de lectura/escritura no se tiene en cuenta, al compilador le da igual lo que contengan esas propiedades, ya que no las evaluará.

En estos ejemplos solo hemos definido una propiedad de solo lectura, pero si los tipos definen más de una propiedad de solo lectura, la igualdad solo se cumplirá si todas y cada una de esas propiedades tienen el mismo valor en ambas instancias.

En el código del listado 4.12 tenemos varias instancias de un mismo tipo anónimo con dos propiedades de solo lectura; al comparar esas instancias se tendrá en cuenta el contenido de las dos para decidir si son iguales o no (recordemos que la propiedad de lectura/escritura no se tiene en cuenta en la comparación). El tipo anónimo de todas las variables es el mismo, sin embargo, la instancia de la variable ***tac6*** utiliza un tipo diferente para la propiedad ***ID***, por tanto, fallará la comparación, ya que el valor *hash* creado para esa propiedad es distinto (el que devuelve el método **GetHashCode**).

```
Dim tac1 = New With {Key .ID = 10, Key .Codigo = "1234", .Precio = 125.5}
Dim tac2 = New With {Key .ID = 99, Key .Codigo = "1234", .Precio = 125.5}
Dim tac3 = New With {Key .ID = 10, Key .Codigo = "1234", .Precio = 888.3}
Dim tac4 = New With {Key .ID = 10, Key .Codigo = "ABCDE", .Precio = 125.5}
Dim tac5 = New With {Key .ID = 10, Key .Codigo = "ABCDE", .Precio = 777.3}
Dim tac6 = New With {Key .ID = "10", Key .Codigo = "1234", .Precio = 125.5}
' False, cambia el valor de ID
sonIguales = tac1.Equals(tac2)

' True, no se evalúan las propiedades de lectura/escritura
sonIguales = tac1.Equals(tac3)

' False, cambia el valor de Codigo
sonIguales = tac1.Equals(tac4)

' False, son del mismo tipo pero las propiedades son de tipo diferente
sonIguales = tac1.Equals(tac6)

' True, no se evalúan las propiedades de lectura/escritura
sonIguales = tac4.Equals(tac5)
```

Listado 4.12. En los tipos con varias propiedades de solo lectura, se tienen en cuenta todas ellas para comparar dos instancias

Si al lector le extraña que todas las variables sean del mismo tipo anónimo, le recuerdo que, en realidad, los tipos anónimos se definen con parámetros *generic* y al crear la instancia es cuando se indica el tipo de datos real que tendrá cada propiedad.

Por supuesto, este comportamiento solo ocurre si los nombres de las propiedades son iguales y están definidos en el mismo orden, ya que ese tipo anónimo, en realidad, se define como vemos en el seu-

docódigo del listado 4.13. La implementación de la clase es más compleja, pero de esta forma vemos los elementos básicos para tener una idea del comportamiento, en particular a la hora de crear la instancia, ya que en el constructor se utilizan los parámetros (propiedades) en el mismo orden en el que se han definido al crear el tipo anónimo. Si tuviésemos acceso a la definición de la clase anónima, las declaraciones del listado 4.12 serían como las del código del listado 4.14.

```
Class VBAnonymousType_4(Of T0, T1, T2)

    Public Sub New(ByVal ID As T0, _
                   ByVal Codigo As T1, _
                   ByVal Precio As T2)
        Me.ID = ID
        Me.Codigo = Codigo
        Me.Precio = Precio
    End Sub

    Public ReadOnly ID As T0
    Public ReadOnly Codigo As T1
    Public Precio As T2
End Class
```

Listado 4.13. Seudocódigo del tipo anónimo creado en el listado 4.12

```
Dim tac1 As New VBAnonymousType_4(Of Integer, String, Double) (10, "1234", 125.5)
Dim tac2 As New VBAnonymousType_4(Of Integer, String, Double) (99, "1234", 125.5)
Dim tac3 As New VBAnonymousType_4(Of Integer, String, Double) (10, "1234", 888.3)
Dim tac4 As New VBAnonymousType_4(Of Integer, String, Double) (10, "ABCDE", 125.5)
Dim tac5 As New VBAnonymousType_4(Of Integer, String, Double) (10, "ABCDE", 777.3)
Dim tac6 As New VBAnonymousType_4(Of String, String, Double) ("10", "1234", 125.5)
```

Listado 4.14. Así se definirían las variables usadas en el listado 4.12 si tuviésemos acceso al tipo anónimo

Independientemente de todo lo que el compilador haga para que podamos usar los tipos anónimos, no cabe duda de que en muchas situaciones nos resultarán de utilidad, particularmente cuando los utilicemos con todo lo relacionado con LINQ.

Pero aún nos quedan otros detalles que debemos conocer sobre los tipos anónimos, tanto para sacarles el máximo rendimiento, como para decidir si utilizar este tipo de clases o usar clases definidas por nosotros.

Los tipos anónimos y la inferencia automática de tipos

Como ya comenté antes, cuando definimos un tipo anónimo, el compilador de Visual Basic utiliza la inferencia automática de tipos para decidir qué tipo de datos tendrán las propiedades e incluso para saber cuál será el tipo anónimo que debe utilizar, (uno existente, si usamos la misma firma o uno nuevo si no existe un tipo anónimo que coincida con el que estamos definiendo).

Pero debemos saber que esa inferencia de tipos la hará el compilador **incluso** si tenemos desactivado **Option Infer**, o casi. Veamos cuál es el "casi".

Si tenemos desactivado **Option Infer**, también debemos desactivar **Option Strict**, ya que de no hacerlo, recibiremos el conocido error del compilador Visual Basic en el que nos avisa de que **Option Strict On** requiere que en todas las declaraciones de las variables usemos la cláusula **As**.

El problema de tener desactivadas estas dos opciones es que el tipo de la variable que recibe la instancia del tipo anónimo es **Object**. Por tanto, IntelliSense no nos mostrará las propiedades a las que podemos acceder, además de que cualquier acceso a esas propiedades se hará en tiempo de ejecución, utilizando lo que se conoce como enlace tardío (*late binding*). Así que, si no queremos padecer de los nervios y facilitar nuestro trabajo, lo mejor es que siempre tengamos activada **Option Strict**, además de **Option Infer**.

> ***Nota***
>
> *Independientemente de cómo tengamos Option Infer, los tipos de datos de las propiedades de los tipos anónimos siempre se inferirán correctamente, donde interviene el estado de Option Infer es en la inferencia de la variable que recibe el objeto del tipo anónimo.*

Los tipos anónimos solo los podemos usar a nivel local

Sabiendo que la inferencia de tipos siempre actúa en la definición de los tipos anónimos, es evidente que su uso solo puede ser a nivel local, es decir, dentro de un procedimiento. Por tanto, no podremos usar los tipos anónimos como parámetro de un método, como valor devuelto por una función o propiedad, ni para definir variables a nivel de módulo (accesible a todo el tipo).

Para que nos sirva de recordatorio, los tipos anónimos solo los podremos definir donde podamos inferir automáticamente los tipos de las variables.

Esto también nos sirve para saber que aunque podamos crear *arrays* o colecciones de tipos anónimos (en realidad Visual Basic no facilita este tipo de creaciones, salvo usando el truco que vimos en el capítulo dos), esas colecciones no podremos utilizarlas fuera del procedimiento en el que lo definimos, salvo que lo almacenemos como **Object**, pero en ese caso perderemos el tipo de los elementos de esa colección o *array*.

En estos casos, en los que necesitamos usar esos objetos de tipo anónimo fuera del procedimiento en el que los definimos (aunque parezca contradictorio), lo mejor es usar tipos que no sean anónimos.

Tipos anónimos que contienen otros tipos anónimos

Como ya comenté, un tipo anónimo lo podemos usar en cualquier sitio en el que podamos inferir una variable, por tanto, también podemos definir tipos anónimos como valor de una propiedad de un tipo anónimo. En el listado 4.15 vemos un ejemplo en el que la propiedad ***Artículo*** es a su vez un tipo anónimo.

```
Dim ta11 = New With {.ID = 1, _
                     .Artículo = New With {.Cantidad = 12}, _
                     .Descripción = "Prueba 11"}
```

Listado 4.15. Un tipo anónimo puede contener otros tipos anónimos

Y si necesitamos que una de esas propiedades sea un *array* o una colección, podemos usar el método ***CrearLista*** que vimos en el capítulo dos. En el listado 4.16 vemos un ejemplo.

```
Dim ta12 = New With {.ID = 2, _
                     .Artículos = _
                            CrearLista(New With {.Cantidad = 6}, _
                                       New With {.Cantidad = 24}, _
                                       New With {.Cantidad = 10}), _
                     .Descripción = "Prueba 12"}
```

Listado 4.16. Incluso las propiedades pueden ser colecciones de tipos anónimos

Ejercicio: ¿de qué tipo será la propiedad ***Artículos***? La respuesta, al final de la siguiente sección.

Recomendaciones

Para finalizar esta primera aproximación a los tipos anónimos, veamos algunas recomendaciones de cómo y cuándo debemos usar los tipos anónimos.

En realidad, los tipos anónimos no están pensados para usarlos como una forma de sustituir los tipos normales, y casi con toda seguridad la mayor utilidad que le veremos a este tipo de datos es cuando los usemos con LINQ, y como de LINQ nos ocuparemos en un próximo capítulo, cuando le llegue el turno comprobaremos que ahí sí que tienen utilidad.

Mientras llega el capítulo de LINQ, repasemos un poco lo que hemos tratado de esta nueva característica de Visual Basic; espero que la siguiente relación le sirva al lector para tener una visión rápida de cómo y cuándo utilizar los tipos anónimos.

- Los tipos anónimos solo los podemos usar a nivel de procedimiento.
- Siempre los crearemos usando la característica conocida como inicialización de objetos.
- Las propiedades pueden ser de solo lectura o de lectura/escritura.
- Las propiedades de solo lectura las definiremos anteponiendo la instrucción **Key**.

- El compilador creará un nuevo tipo anónimo si no existe uno a nivel de ensamblado que tenga la misma firma.
- En la firma de un tipo anónimo se tendrá en cuenta el número de propiedades, los tipos, los nombres (no se diferencia entre mayúsculas y minúsculas) y si son de solo lectura o de lectura/escritura.
- Solo podemos comparar la igualdad de dos instancias de tipos anónimos si son del mismo tipo y definen propiedades de solo lectura, en ese caso, la igualdad se comprobará teniendo en cuenta todas las propiedades de solo lectura.

Solución al ejercicio del listado 4.16

Si usamos la definición del método ***CrearLista*** del listado 2.9 del segundo capítulo, la propiedad ***Artículos*** será del tipo **IEnumerable**(**Of** ***el_tipo_anónimo***), ya que el método **CrearLista** (con parámetros *generic*) infiere el tipo de datos que estamos usando como *array* de parámetros opcionales, es decir, el tipo anónimo que el compilador utilice para cada uno de los argumentos de esa función.

> ***Versiones***
>
> *Esta característica solo la podemos usar con .NET Framework 3.5 y no es necesario añadir ninguna importación especial.*

Expresiones lambda

Según la documentación de Visual Studio 2008, una expresión *lambda* es una función sin nombre que podemos usar en cualquier sitio en el que un delegado sea válido.

El lector que conozca algo de C# seguramente esta definición le recordará a los métodos anónimos que introdujo ese lenguaje en la versión 2.0, pues algo parecido, pero no exactamente lo mismo.

En Visual Basic 9.0 las expresiones *lambda* (o funciones anónimas o funciones en línea) son funciones que solo pueden tener una instrucción que será la encargada de devolver el valor. Esto significa que, en realidad, no es una función como las que estamos acostumbrados a definir en Visual Basic, entre otras cosas, por el hecho de que solo puede tener una instrucción, y cuando digo una instrucción me refiero a una sola instrucción, no a una sola línea de código (que también debe estar esa instrucción en una sola línea de código, aunque en esa línea de código podemos usar el guión bajo, pero solo para facilitar la lectura). Este comentario viene a cuento de que en Visual Basic podemos escribir en una misma línea varias instrucciones separadas por dos puntos.

En realidad, el contenido de las funciones anónimas se reduce a una sola expresión que devuelve un valor, de ahí que no podamos escribir más de una instrucción. Sabiendo esto, después veremos algunos trucos para ampliar el contenido de una función anónima.

> ***Nota***
>
> *A lo largo de este texto (y lo que resta del libro) usaré los términos "expresiones lambda", "funciones anónimas" o "funciones en línea" para referirme a esta característica de Visual Basic 9.0. El primero es el término usado en la documentación de Visual Studio 2008, y que desde mi punto de vista es más apropiado para los que gustan de usar C# como lenguaje de programación, ya que el operador usado para las expresiones lambda de C# se llama "operador lambda" (=>); el segundo es porque en Visual Basic, en realidad, se definen como una función que no tiene nombre; y el tercero porque esa función la podremos usar directamente en el código, sin necesidad de crearla de forma independiente*.

Entendiendo a los delegados

Como ya he comentado, las expresiones *lambda* se pueden usar en cualquier lugar en el que podamos usar un delegado, pero esto necesita de un poco de clarificación para comprender mejor esta característica.

Un delegado define la firma que debe tener el método al que queremos invocar desde un objeto del tipo de ese delegado. El caso habitual en el que se usan los delegados es en los eventos, una clase define un evento (que en realidad es una variable del tipo de un delegado) y cuando quiere informar que ese evento se ha producido, hace una llamada a cada uno de los métodos que se han suscrito a ese evento. En Visual Basic, para indicar el método que queremos suscribir a un evento, utilizamos la instrucción **AddressOf** seguida del nombre del método. Por ejemplo, si tenemos una aplicación de **Windows.Forms** y queremos interceptar el evento **Click** de un botón llamado ***Button1***, tendremos que definir un método que tenga la misma firma que el delegado asociado a ese evento, y para asociar ese evento con ese método lo haremos tal como vemos en el listado 4.17.

```
AddHandler Button1.Click, AddressOf Button1_Click
```

Listado 4.17. AddHandler requiere un delegado en el segundo parámetro

El segundo parámetro de la instrucción **AddHandler** espera un delegado, éste indicará la dirección de memoria del método al que se debe llamar cuando se produzca el evento indicado en el primer parámetro.

En Visual Basic, para indicar la dirección de memoria de un método, tenemos que usar la instrucción **AddressOf** seguida del nombre del método. Y, como vemos en el listado 4.17, esa dirección de memoria la podemos indicar directamente.

Aunque, en realidad, el compilador lo que hace es crear una instancia del tipo del delegado y pasarle la dirección de memoria del método. Si quisiéramos hacer un poco más manual todo este proceso, el código del listado 4.17 lo podríamos sustituir por el del listado 4.18.

```
AddHandler Button1.Click, New EventHandler(AddressOf Button1_Click)
```

Listado 4.18. El equivalente del listado 4.17 usado por Visual Basic

La definición del listado 4.18 la podemos escribir también de la forma mostrada en el listado 4.19, en el que definimos una variable del mismo tipo del delegado que queremos usar con la instrucción **AddHandler**, ese delegado lo pasamos como segundo argumento de la instrucción.

```
Dim delegado As New EventHandler(AddressOf Button1_Click)
AddHandler Button1.Click, delegado
```

Listado 4.19. Los delegados los podemos asignar a variables

En realidad, en Visual Basic todo esto es mucho más sencillo, pero en el fondo es lo que el compilador hace, y creo que nos servirá para comprender mejor la parte de la definición de las expresiones *lambda* en la que nos indica que las "*podemos usar en cualquier sitio en el que un delegado sea válido*".

Definir una expresión lambda

Las expresiones *lambda* las tenemos que definir usando la instrucción **Function** seguida de los parámetros que se usarán en la función y después de los parámetros usaremos una expresión; el tipo de datos que devuelva esa expresión será el tipo de datos de la función.

Al definir las expresiones *lambda* no indicaremos ni el tipo de datos que devuelve la función ni usaremos la instrucción **Return** para devolver ese valor.

Por ejemplo, si queremos sumar dos valores enteros y devolver el resultado de la suma, lo podemos hacer tal como vemos en el listado 4.20.

```
Dim delSuma = Function(n1 As Integer, n2 As Integer) n1 + n2
```

Listado 4.20. Definición de una expresión lambda

Para usar la función anónima del listado 4.20 lo haremos por medio de la variable ***delSuma***, que para usarla tendremos que indicar los dos valores que queremos sumar y el resultado lo podemos guardar en una variable, usarlo en una expresión o lo que creamos conveniente. En el listado 4.21 tenemos un ejemplo de uso de ese delegado.

```
Dim i As Integer = delSuma(15, 22)
```

Listado 4.21. Uso del delegado que contiene una función anónima

El compilador en realidad lo que hace al encontrarse con la definición de una expresión *lambda*, es crear un delegado *generic* con esa misma firma, es decir, una función que recibe dos valores. El seudocódigo generado por Visual Basic sería parecido al mostrado en el listado 4.22.

```
Delegate Function _
            VBAnonymousDelegate_0(Of T0, T1, T2)(ByVal n1 As T0, _
                                                 ByVal n2 As T1 _
                                                 ) As T2
```

Listado 4.22. Definición del delegado para la expresión lambda del listado 4.20

También crea una función que es la que se encarga de hacer la suma de los dos parámetros pasados a la expresión *lambda*; el seudocódigo sería como el del listado 4.23.

```
Private Function _Lambda_1(ByVal n1 As Integer, _
                           ByVal n2 As Integer) As Integer
    Return n1 + n2
End Function
```

Listado 4.23. La función lambda creada por Visual Basic con la firma del delegado

La variable ***delSuma*** quedaría definida como vemos en el listado 4.24, y la forma de usar ese delegado será igual que el listado 4.21, eso no cambia.

```
Dim delSuma As New  _
            VBAnonymousDelegate_0(Of Integer, Integer, Integer) _
                                        (AddressOf _Lambda_1)
```

Listado 4.24. Definición de la variable del tipo del delegado

Por supuesto, para utilizar las expresiones *lambda* no tenemos por qué saber todos estos pormenores, pero así tendremos una visión más clara de lo que tendríamos que hacer, si no existiera este tipo de funciones anónimas.

Las expresiones lambda son funciones en línea

El código mostrado en los listados 4.20 y 4.21 lo podemos resumir en algo más conciso, de forma que no sea necesario crear un delegado intermedio; el código del listado 4.25 sería equivalente a esos dos.

```
Dim k = (Function(x As Integer, y As Integer) x + y)(15, 22)
```

Listado 4.25. Una expresión lambda usada directamente

En el ejemplo del listado 4.25, la variable ***k*** recibe el valor resultante de evaluar la expresión, es decir, es de tipo **Integer**, ya que en este caso no creamos ninguna variable intermedia para almacenar la función que realiza la operación. Por supuesto, el código que genera el compilador es parecido a todo lo que vimos anteriormente.

Como podemos comprobar, el código del listado 4.25 no tiene mucha utilidad ya que la variable ***k*** recibe el cálculo que hace la expresión *lambda* y para hacer esa suma no es necesario complicarse tanto, pero más que nada es para ver lo que podemos hacer con este tipo de funciones anónimas.

Las expresiones lambda como parámetro de un método

Cuando queremos pasar por argumento a un método otro método, el parámetro que recibirá la dirección de memoria de ese método debe ser un delegado y como las expresiones *lambda* las podemos usar en cualquier sitio en el que se pueda usar un delegado, también podremos usar las funciones anónimas para realizar esta tarea.

Para comprenderlo mejor, veamos primero cómo lo haríamos con delegados pero al estilo de la versión anterior de Visual Basic, de esta forma tendremos más claro todo lo relacionado a las expresiones *lambda* y la facilidad que tenemos ahora para hacer ciertas tareas que antes suponía tener que escribir más código y hacer más pasos intermedios.

El ejemplo consiste en definir un método que reciba dos valores de tipo entero y un tercer parámetro que será un delegado. Éste lo definimos como una función que recibe dos parámetros de tipo entero y devuelve otro entero. Para llamar a este método, tendremos que indicar dos números y la dirección de memoria de una función con la misma firma del delegado; esa función simplemente sumará los dos valores.

En los siguientes listados tenemos el código que hace todo esto que acabo de comentar.

```
Delegate Function SumaCallback(ByVal num1 As Integer, _
                               ByVal num2 As Integer) As Integer
```

Listado 4.26. Definición del delegado

```
Function sumar(ByVal num1 As Integer, _
               ByVal num2 As Integer) As Integer
    Return num1 + num2
End Function
```

Listado 4.27. La función con la misma firma que el delegado

```
Sub conDelegado(ByVal x As Integer, _
                ByVal y As Integer, _
                ByVal d As SumaCallback)
    Console.WriteLine("x = {0}, y = {1}", x, y)
    Console.WriteLine("d(x,y) = {0}", d(x, y))
End Sub
```

Listado 4.28. Un método que recibe un delegado como parámetro

```
conDelegado(10, 20, AddressOf sumar)
```

Listado 4.29. Una de las formas de usar el método del listado 4.28

Para usar el método del listado 4.28 tenemos que pasarle dos valores de tipo entero y la dirección de memoria del método que se usará cuando se llame al delegado. La forma más sencilla de hacerlo es tal como vemos en el listado 4.29, pero también podríamos crear una variable del tipo del delegado y usar esa variable como argumento del tercer parámetro, tal como vemos en el listado 4.30.

```
Dim sumaDel As SumaCallback = AddressOf sumar
conDelegado(22, 33, sumaDel)
```

Listado 4.30. Otra forma de utilizar el método del listado 4.28

El método ***conDelegado*** espera recibir un delegado en el tercer parámetro, y como sabemos, las expresiones *lambda*... ¡efectivamente! *las podemos usar en cualquier sitio en el que se pueda usar un delegado*, por tanto, el código del listado 4.31 también sería válido.

```
conDelegado(30, 70, Function(x, y) x + y)
```

Listado 4.31. El método del listado 4.28 lo podemos usar con una expresión lambda

La ventaja de usar el código del listado 4.31 es que nos libra de tener que definir una función que tenga la firma del delegado, ya que al usar la función anónima obtenemos el mismo resultado.

Otra ventaja es que podríamos querer usar ese mismo método con otra función que, en lugar de sumar los parámetros hiciera otra operación, con las funciones anónimas sería una tarea fácil, ya que

solo tendríamos que cambiar la expresión que devuelve el valor, como en el listado 4.32 en el que efectuamos una resta.

```
conDelegado(30, 70, Function(x, y) x - y)
```

Listado 4.32. Al ser una función en línea, podemos cambiar el resultado a devolver

Si esto mismo quisiéramos hacerlo en Visual Basic 8.0/2005 nos veríamos obligados a definir otra función que hiciera esa operación de restar los dos valores y llamar a este método pasándole la dirección de memoria de esa función.

En .NET Framework 3.5 hay una serie de delegados *generic* que nos pueden servir para reducir aún más nuestro código. Como hemos visto en los listados 4.26 y 4.28, para crear ese método que recibe un delegado, antes hemos tenido que definir el delegado (también declaramos un método -listado 4.27- para usar al llamar al método del listado 4.26, pero con las expresiones *lambda*, esa definición nos la ahorramos), con los delegados que .NET 3.5 define, también nos podemos ahorrar el delegado. Por supuesto, si no definimos el delegado del listado 4.26 tendremos que cambiar la definición del método mostrado en el listado 4.28. La nueva definición será como la mostrada en el listado 4.33. **Func** es un delegado *generic* definido en .NET Framework 3.5, en este ejemplo, utiliza tres tipos de datos, los dos primeros representan los parámetros de la función y el tercero es el valor que devuelve esa función. Para usar ese método, lo haremos como vemos en el código del listado 4.34, la operación que realizamos en este caso es una multiplicación.

```
Sub conDelegado2(ByVal x As Integer, _
                 ByVal y As Integer, _
                 ByVal d As Func(Of Integer, Integer, Integer))
    Console.WriteLine("x = {0}, y = {1}", x, y)
    Console.WriteLine("d(x,y) = {0}", d(x, y))
End Sub
```

Listado 4.33. Nueva definición del método que recibe un delegado en el tercer parámetro

```
conDelegado2(50, 20, Function(x, y) x * y)
```

Listado 4.34. Llamada al método que define el delegado Func

.NET 3.5 define 5 sobrecargas del delegado *generic* **Func** según el número de parámetros que se utilicen (usando siempre el último para indicar el valor devuelto), todos ellos de tipo **Function**, ya que siempre devuelven algo; si necesitamos un delegado que no devuelva nada (equivalente a un procedimiento **Sub**), podemos usar el delegado *generic* **Action**, que al igual que **Func** tiene 5 sobrecargas según los parámetros que reciba.

Todos estos delegados, al definir los parámetros de tipo *generic*, se pueden usar para cualquier tipo de datos, en el ejemplo del listado 4.33, los tres parámetros usados son de tipo **Integer**.

Volvamos a ver el código del listado 4.31 (o del listado 4.32 o del listado 4.34). ¿Qué nota en esa forma de usar la expresión *lambda* del tercer parámetro?

Si miramos la diferencia de los tres listados, puede que nos parezca que sea la expresión usada para devolver el valor, pero no es eso lo que debemos notar, ya que lo extraño en esas expresiones *lambda* es que no hemos indicado los tipos de datos de las dos variables usadas en la función anónima. En este caso, entra en funcionamiento la inferencia automática de tipos, y como el compilador sabe que el delegado que se utiliza para el tercer parámetro tiene dos parámetros de tipo **Integer**, asume que esas dos variables deben ser de ese tipo o de cualquier tipo que se pueda convertir de forma implícita al tipo **Integer**.

Pero debemos tener en cuenta que si no indicamos los tipos de datos, el compilador los infiere según la definición del delegado, pero si indicamos el tipo, ese tipo debe ser del "adecuado", es decir, de tipo **Integer**. Además de que si indicamos el tipo de uno de los parámetros, debemos indicar el tipo de todos ellos; sabiendo esto, el código del listado 4.35 daría error, porque solo indicamos el tipo de datos del primer parámetro.

```
conDelegado2(25, 71, Function(x As Integer, y) x + y)
```

Listado 4.35. Si indicamos el tipo de un parámetro, debemos indicarlos todos

Ámbito en las expresiones lambda

Viendo el código de los ejemplos mostrados hasta ahora es posible que nos llevemos la impresión de que en la expresión de una función anónima siempre usaremos las variables indicadas en los parámetros.

Pero eso no es así, en esa expresión podemos usar cualquier constante o variable que esté en el mismo ámbito de la función anónima, es decir, además de los parámetros, también podemos usar las variables que estén definidas en el mismo procedimiento en el que estemos usando la expresión *lambda* o cualquier otra variable definida en cualquier otra parte, pero siempre que esté accesible, por ejemplo, las variables definidas a nivel de tipo.

En esto no cambia con respecto a las funciones normales, la diferencia es que las expresiones *lambda* siempre las usaremos desde "dentro" de un procedimiento, por tanto, todas las variables locales de ese procedimiento también las podremos utilizar para evaluar la expresión que devolverá.

Lo que debemos tener en cuenta es que el "cuerpo" de una función anónima siempre será una expresión y debemos imaginarnos que delante de esa expresión hay una instrucción **Return**, de esta forma nos resultará más fácil saber qué es lo que puede contener la expresión.

En el listado 4.36 vemos un ejemplo en el usamos variables definidas fuera de la expresión *lambda*.


```
Dim i1 As Integer = 10
Dim d1 As Double = 22.5

Dim d = Function(n As Integer) (n * i1) + d1

Console.WriteLine(d(12))
```

Listado 4.36. En las expresiones lambda podemos usar variables que estén en ámbito

Si usamos variables externas dentro de la expresión de una función anónima, y esas variables tienen una vida más corta que la propia función, no habrá problemas de que el recolector de basura las elimine cuando pierdan el ámbito, ya que al estar siendo usadas, la vida de las mismas se mantiene.

Por ejemplo, en el listado 4.37 tenemos la definición de un método que devuelve un delegado como el que hemos estado usando en los listados anteriores. El valor devuelto por ese método es una expresión *lambda*, que utiliza una variable local (***k***) para efectuar el cálculo que debe devolver cuando se utilice.

```
Function conDelegado3(ByVal x As Integer, _
                      ByVal y As Integer _
                      ) As Func(Of Integer, Integer, Integer)
    Dim k = (x + y) \ 2
    Console.WriteLine("k = {0}", k)

    Return Function(n1 As Integer, n2 As Integer) n1 * k + n2 \ k
End Function
```

Listado 4.37. Un método que devuelve una expresión lambda

En el código del listado 4.38 usamos ese método, y guardamos una referencia a la función anónima creada dentro del método, después usamos en varias ocasiones esa variable, y como podemos imaginar, cada vez que utilizamos esa variable estamos invocando a la función anónima, que aún sigue estando activa y conservando el valor que tenía la variable ***k***. Ese valor no se perderá aunque el método (***conDelegado3***) ya haya terminado y, por tanto, todo el contenido del mismo se haya desechado. Si nos sirve de comparación, esto es similar a lo que ocurre cuando creamos un formulario dentro de un método y aunque el método finalice, el formulario sigue estando operativo hasta que finalmente lo cerremos.

```
Dim d2 = conDelegado3(7, 4)
Dim s = d2(3, 3)
Console.WriteLine(s)

s = d2(5, 7)
Console.WriteLine(s)
```

Listado 4.38. Mientras el delegado devuelto por la función siga activo, la variable local usada para efectuar los cálculos estará “viva”

Otro ejemplo parecido es si usamos una expresión *lambda* para asignarla a un manejador de eventos, es decir, usar una función anónima en lugar de un método.

Utilizar una expresión lambda como un método de evento

Si retomamos el código que usamos al principio para asignar el método del evento **Click** de un botón, podríamos definirlo como una expresión *lambda* en lugar de indicar un método existente. En el listado 4.39 vemos cómo definir este tipo de función anónima, que la podemos hacer en el evento **Load** del formulario (o en cualquier otro sitio, pero antes de usar ese evento). El contenido de la expresión utiliza dos controles que tiene ese formulario, y como hemos comprobado, cuando llegue el momento (en este caso, cuando el usuario haga clic en ese botón) se evaluará esa expresión, por tanto, en cada una de esas invocaciones se añadirá al contenido de la colección **Items** del **ListBox** el texto que tenga el control ***TextBox1***.

```
AddHandler btnAdd.Click, Function() ListBox1.Items.Add(TextBox1.Text)
```

Listado 4.39. Utilizar un método de evento por medio de una expresión lambda

En la definición de la expresión *lambda* del listado 4.39 nos aprovechamos de la relajación en las conversiones de los delegados para no tener en cuenta los dos parámetros que el delegado asociado al evento **Click** define.

Ampliando la funcionalidad de las expresiones lambda

Sabiendo que en las expresiones de las funciones anónimas podemos usar cualquier elemento que esté definido en el código (siempre que esté en ámbito), caeremos en la cuenta de que si podemos usar variables u objetos de cualquier tipo, también podremos usar otras funciones, siempre y cuando tengamos en cuenta que esa expresión es solo una "simple" expresión, lo de *simple* es para enfatizar que es solamente una expresión, pero que podemos complicar todo lo que queramos, incluso utilizando otras funciones anónimas como parte de esa expresión.

En el listado 4.40 tenemos la definición de dos funciones anónimas donde la segunda utiliza la primera para realizar sus cálculos. En el listado 4.41 tenemos el equivalente en el que usamos dos expresiones *lambda*. En el listado 4.41 he tenido que cambiar el nombre de la primera variable usada como parámetro, ya que si dejara el mismo nombre que la que defino después en la segunda función anónima, el compilador daría un error de que ese nombre ya se está usando, error que es lógico, ya que ambas funciones anónimas están en una misma expresión.


```
Dim d1 = Function(x As Integer, y As Integer) x + y
Dim d2 = Function(x As Integer) x * d1(x, 2)

Console.WriteLine(d2(5))
```

Listado 4.40. Definición de dos funciones anónimas en la segunda se utiliza la primera

```
Dim d3 = Function(n As Integer) n * _
             (Function(x As Integer, y As Integer) n + y)(n, 2)

Console.WriteLine(d3(5))
```

Listado 4.41. Código equivalente al código del listado 4.40 pero anidando las dos funciones anónimas

Por supuesto, además de usar funciones anónimas, también podemos usar funciones normales, es decir, cualquier cosa que podamos escribir en una expresión la podemos escribir en la expresión que devuelve el valor de una función anónima.

¿Cómo clasificar una colección de tipos anónimos?

Como vimos al hablar de los tipos anónimos, podemos usar la función *generic* ***CrearLista*** para asignar una colección de tipos anónimos a una propiedad de otro tipo anónimo, si modificamos esa función para que devuelva una colección de tipo **List(Of T)**, podríamos clasificar esa colección, ya que esa clase tiene un método **Sort**.

Como sabemos, para poder clasificar los elementos de una colección, los objetos almacenados en la colección deben implementar la interfaz **IComparable**, y las clases anónimas no implementan esa interfaz, además de que aunque pudiéramos implementarla, tampoco podríamos escribir un método para que realice las comparaciones, ya que los tipos anónimos no nos permiten definir métodos.

Pero todos los métodos de clasificación permiten que indiquemos una clase basada en **IComparer** que sea la que se encargue de realizar las comparaciones de los elementos que no implementan directamente la interfaz que define el método **CompareTo**. El problema es que esa clase necesita conocer los tipos de datos con los que debe trabajar, por tanto, no sería la solución, ya que, como vimos anteriormente, no tenemos forma de acceder al tipo interno que el compilador usa para los tipos anónimos.

Una de las sobrecargas del método **Sort** de la clase genérica **List(Of T)** permite que indiquemos un método que sea el encargado de comparar dos elementos de la colección y devolver el valor adecuado a la comparación. Nuevamente nos encontramos con el problema de que, para que ese método funcione, la comparación se debe hacer con dos elementos del tipo que queremos clasificar, y si esos tipos son anónimos ¿cómo los indicamos? Y aquí es donde entran en escena las extensiones de .NET Framework 3.5, particularmente las funciones anónimas y la relajación de delegados, ya que ese método que espera **Sort**, lo podemos indicar como una expresión *lambda*, de forma que reciba dos objetos de los contenidos en la colección y hagamos la comparación que creamos conveniente. Mejor lo vemos con un ejemplo.

En el listado 4.42 tenemos una adaptación del código mostrado en el listado 4.16, en el que creamos un tipo anónimo, que a su vez contiene una colección de tipos anónimos, y para clasificar esa colección usaremos el código del listado 4.43 en el que usamos una función en línea para realizar la comparación de dos elementos. En este ejemplo tenemos dos propiedades en los elementos "anónimos" de la colección, pero usaremos solo una de esas propiedades para realizar la clasificación.

```
Dim ta13 = New With {.ID = 17, _
                     .Artículos = _
                            CrearLista(New With { _
                                        .ID = 95, .Cantidad = 6}, _
                                       New With { _
                                        .ID = 22, .Cantidad = 24}, _
                                       New With { _
                                        .ID = 95, .Cantidad = 1}, _
                                       New With { _
                                        .ID = 39, .Cantidad = 10} _
                                      ), _
                     .Descripción = "Prueba 13"}
```

Listado 4.42. Un tipo anónimo que tiene una colección de tipos anónimos

```
ta13.Artículos.Sort(Function(x, y) x.ID.CompareTo(y.ID))
```

Listado 4.43. Usamos una expresión lambda como función a usar con el método Sort

En el código del listado 4.43, la sobrecarga que usamos del método **Sort** es la que utiliza el delegado **Comparison(Of T)**, éste recibe dos parámetros, los compara y devuelve un valor cero si son iguales, menor de cero si el primer argumento es menor que el segundo, o mayor de cero si el segundo es mayor (es decir, los valores típicos para las comparaciones). En nuestro ejemplo, el delegado (el método que hará las comparaciones), es una función *lambda* que recibe dos objetos, y gracias a la inferencia automática de tipos, no es necesario indicar el tipo de los mismos, ya que el compilador sabe que son del tipo anónimo que hemos almacenado en esa colección, por eso nos permite acceder a las propiedades de esos parámetros, que en este ejemplo hemos usado el método **CompareTo** de la propiedad ***ID***. Ese método anónimo se usará cada vez que el *runtime* de .NET necesite comparar dos elementos de la colección, es decir, clasificará el contenido de la colección ***Artículos*** por el valor de la propiedad ***ID***.

En el listado 4.42, cada uno de los elementos de la colección ***Artículos*** es de tipo anónimo con dos propiedades en lugar de una, que es como lo teníamos en el listado 4.16, pero esto solo lo he hecho para usar un código diferente, no porque sea necesario que el tipo de datos a comparar tenga una propiedad llamada ***ID***.

Como comenté al principio, el método ***CrearLista*** tiene que devolver un objeto del tipo **List(Of T)**, por tanto, debemos cambiar la definición de ese método de forma que usemos la del listado 4.44, que como vemos es prácticamente la misma que la mostrada en el capítulo dos.


```
Function CrearLista(Of T)(ByVal ParamArray datos() As T) As List(Of T)
    Dim lista As New List(Of T)
    lista.AddRange(datos)
    Return lista
End Function
```

Listado 4.44. Definición de la función CrearLista para que devuelva un tipo que define un método Sort

En la comparación que hacemos en el cuerpo de la función anónima podemos comparar lo que queramos, no solo los valores solitarios de las propiedades de esa clase anónima. Por ejemplo, en el listado 4.45 tenemos una comparación algo más complicada, esto es posible porque esa función debe devolver un valor entero indicando el orden (0, 0< o >0) y la forma de conseguir ese valor no le preocupa al CLR de .NET.

```
ta13.Artículos.Sort(Function(x, y) _
                      (x.ID.ToString("00") & x.Cantidad.ToString("000")) _
                      .CompareTo _
                      (y.ID.ToString("00") & y.Cantidad.ToString("000")) _
                    )
```

Listado 4.45. Modificación de la función anónima para clasificar los elementos usando las dos propiedades del tipo anónimo

Como sabemos, en la expresión usada en el cuerpo de la función anónima, podemos usar cualquier código que sea válido en una expresión y siempre que devuelva un valor del tipo esperado (**Integer** en nuestro ejemplo), por tanto, podríamos crear una función con nombre que sea la encargada de hacer las comprobaciones necesarias para clasificar esos dos elementos. En el listado 4.46 tenemos el equivalente al listado 4.45, pero usando una función externa, que es la definida en el listado 4.47.

```
ta13.Artículos.Sort(Function(x, y) _
                        comparaEnteros( _
                            x.ID, x.Cantidad, _
                            y.ID, y.Cantidad) _
                    )
```

Listado 4.46. La expresión usada en la función anónima usa una función externa

```
Function comparaEnteros(ByVal x1 As Integer, ByVal x2 As Integer, _
                        ByVal y1 As Integer, ByVal y2 As Integer _
                        ) As Integer
    Dim x, y As String
    x = x1.ToString("00") & x2.ToString("000")
    y = y1.ToString("00") & y2.ToString("000")
    Return x.CompareTo(y)
End Function
```

Listado 4.47. La función usada por la expresión lambda del listado 4.46

En los listados 4.45 y 4.47 realizamos una comparación de forma que se clasifiquen los elementos por el valor de la propiedad ***ID*** teniendo en cuenta el valor de la cantidad, de forma que si hay dos elementos con el mismo ***ID*** se muestren clasificados por la cantidad además del identificador. Usando los elementos del listado 4.42, y mostrándolos como vemos en el listado 4.48, el resultado sería el mostrado en la figura 4.2.

```
Console.WriteLine("{0}, {1}", ta13.ID, ta13.Descripción)
For Each a In ta13.Artículos
    Console.WriteLine("  {0}", a.ToString)
Next
```

Listado 4.48. Mostrar el contenido del tipo anónimo y de la colección Artículos

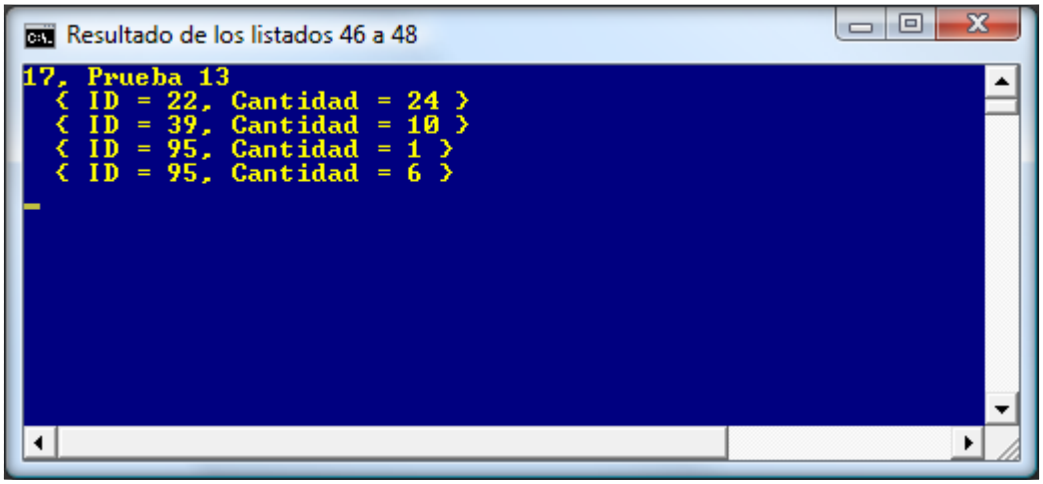

Figura 4.2. Resultado de mostrar los datos usando el listado 4.48

Para realizar todas estas comparaciones, lo ideal es que tuviésemos acceso al tipo anónimo usado como elemento de la colección ***Artículos*** y de esa forma poder pasar esos dos elementos a una función que se encargue de hacer las comparaciones oportunas, pero accediendo a las propiedades directamente. Esto no es posible, ya que no podemos acceder a los tipos anónimos; si pudiéramos, dejarían de ser anónimos.

Me va a permitir el lector que mire a otro lado, porque no quiero que se me relacione con lo que voy a decir a continuación.

Por suerte, Visual Basic nos permite hacer las cosas de forma "no estricta", y en esta ocasión puede estar un poco justificado prescindir de **Option Strict On**. Para entender mejor este comentario, veamos el código del listado 4.49, en el que usamos una expresión *lambda* como argumento del método **Sort**. En esta ocasión usamos un método externo al que le pasamos los dos objetos (del tipo anónimo almacenado en la colección **Artículos**). Como sabemos, esto no se puede hacer si tenemos activada la comprobación estricta del código, ya que esa función debe definir los parámetros del tipo

adecuado, en este caso del tipo anónimo usado en la colección. No quisiera parecer repetitivo, pero a estas alturas ya sabemos que no podemos definir parámetros de un tipo anónimo.

```
ta13.Artículos.Sort(Function(x, y) Module3.comparaArticulos(x, y))
```

Listado 4.49. Clasificamos los elementos de la colección Artículos pasándole dos objetos del tipo anónimo a una función definida en otro módulo

Pero si desactivamos **Option Strict** podemos engañar al compilador y definir ese método para que reciba dos valores de tipo **Object**, y después usar la compilación tardía (*late binding*) para acceder a las dos propiedades que nosotros sabemos que tiene ese tipo anónimo, y eso es lo que hace el código del listado 4.50.

```
Option Strict Off
Option Infer On

Imports System

Module Module3
    Function comparaArticulos(ByVal x As Object, _
                              ByVal y As Object) As Integer
        Dim a1 = x.ID.ToString("00") & x.Cantidad.ToString("000")
        Dim a2 = y.ID.ToString("00") & y.Cantidad.ToString("000")
        Return a1.CompareTo(a2)
    End Function
End Module
```

Listado 4.50. Una función que utiliza late binding para acceder a las propiedades de dos objetos

Ni qué decir tiene que el compilador no comprueba (entre otras cosas, porque no puede) si esos objetos definen esas dos propiedades que usamos, por tanto, hasta que no se ejecute ese código no se sabrá si es correcto o no. En nuestro ejemplo, ese código funcionará, pero ya sabemos que si el tipo de datos que pasamos como argumento de esa función no definiera esas dos propiedades, recibiríamos una excepción en tiempo de ejecución, echando al traste la aplicación.

Moraleja: Si necesitamos hacer cosas que dependan de ciertas propiedades de los elementos de una colección o cualquier otro tipo de datos, es preferible usar tipos "normales" y dejar los anónimos para cuando no tengamos otra posibilidad.

Consideraciones para las expresiones lambda

Para finalizar este tema de las expresiones *lambda* (al menos hasta que veamos todo lo referente a LINQ, ya que este tipo de funciones se utilizan bastante con esa tecnología), repasemos un poco la forma de definir y usar este tipo de funciones anónimas.

En la siguiente lista vemos una serie de condiciones o temas que debemos tener en cuenta a la hora de definir una expresión *lambda*.

- Las expresiones *lambda* se definen usando la instrucción **Function**.
- A diferencia de los procedimientos de tipo **Function**, no se indica un nombre después de esa instrucción.
- No se indica tampoco el tipo de datos que devuelve, ese tipo siempre se infiere dependiendo del valor devuelto en la expresión.
- Para devolver el valor no se utiliza la instrucción **Return**. El valor devuelto es la expresión que indicamos en el cuerpo de la función.
- Solo podemos usar una expresión como valor a devolver, aunque esa expresión puede contener otras funciones anónimas anidadas e incluso usar funciones y variables externas a las indicadas como parámetro.
- Debido a que solo se usa una expresión como cuerpo de la función, no podemos indicar el final de la función (**End Function**).
- Las expresiones *lambda* se pueden usar con parámetros, si no se pueden inferir los tipos de esos parámetros, tenemos que indicarlos expresamente.
- Si se indica el tipo de uno de los parámetros, debemos indicarlos todos, independientemente de que se pueda inferir el tipo.
- Los parámetros de la expresión *lambda* no pueden ser opcionales, por tanto, no podemos usar **Optional** ni **ParamArray**.
- No podemos indicar parámetros *generic*.
- Los nombres de los parámetros no pueden ser los mismos que los de otros elementos que estén en el mismo ámbito.
- Podemos usar una expresión *lambda* como valor devuelto por una función con nombre, pero el tipo del valor devuelto por ésta debe ser del tipo de un delegado.
- Las funciones anónimas las podemos usar para cualquier tipo de delegado, ya sean definidos por nosotros o los que el propio .NET define.

En los próximos capítulos veremos otros usos de este tipo de funciones, particularmente en otros contextos, como el que nos permiten las expresiones de consulta (LINQ).

Versiones

Esta característica solo la podemos usar con .NET Framework 3.5 y no es necesario añadir ninguna importación especial.