Guillermo "Guille" Som

Delegados y eventos

Primera parte: ¿En quién delegas tú?

En este número vamos a tratar de los delegados, y también de los eventos (aunque de estos últimos nos encargaremos con más detalle en el próximo artículo), ya que en .NET están estrechamente relacionados; tanto es así que no podemos definir un evento sin la intervención de un delegado.

» Delegados y eventos en Visual Basic y C#

Seguramente los programadores de Visual Basic estarán pensando que lo dicho en la entradilla no es totalmente cierto. Bueno, posiblemente no, porque si están leyendo **dotNetManía** sabrán lo que se esconde tras los eventos. Pero no está de más aclararlo para que no queden dudas.

Lo cierto es que Visual Basic es un lenguaje muy protector, y nos "libera" de ciertas tareas para facilitarnos el trabajo real, de forma que nos podamos concentrar en lo que de verdad importa y olvidarnos un poco de ciertos "asuntillos" que en parte solo nos hacen teclear más.

Aclaremos un poco todo esto que acabo de comentar. En Visual Basic, para definir un evento solo hay que escribir la instrucción `Event` seguida de la definición que daremos al método que recibirá los eventos. Por ejemplo, si nuestra clase es del tipo `Button`, podemos definir el evento `Click` de la siguiente forma:

```
Public Event Click( ByVal sender As Object,_
                    ByVal e As EventArgs )
```

Como podemos comprobar, esa es la definición que usaremos en cualquier formulario que quiera interceptar la pulsación en un botón. Simple, ¿verdad? Veamos ahora cómo tendría que definir ese mismo evento un programador de C#:

```
public delegate void ClickEventHandler(
                        object sender, EventArgs e);
public event ClickEventHandler Click;
```

En este caso, primero se define un delegado y a continuación hay que definir el evento que debe ser del tipo de ese delegado. Complicado, ¿verdad? Ahora mismo no entraremos en muchos detalles sobre esto, antes veamos cómo se lanzaría ese evento tanto desde Visual Basic como desde C#:

```
' En Visual Basic:
RaiseEvent Click(sender, e)

// En C#:
if( Click != null )
{
    Click(sender, e);
}
```

Indudablemente en Visual Basic sigue siendo mucho más fácil, más simple, menos complicado, no tenemos que comprobar nada... Y es cierto, a eso es a lo que me refería con lo de que Visual Basic es muy "protector" y nos libera de ciertos detalles que en realidad no necesitamos saber, o al menos, no es obligatorio que sepamos. Por otra parte a los programadores de C#, seguramente por aquello de que les gusta "escribir más" código, pues... ¡que escriban más! Aunque, como veremos en este artículo, eso ya está cambiando, y ahora podrán hacer también ciertas cosas sin necesidad de escribir tanto, ya que el propio compilador de C# se encargará de algunos aspectos, digamos, de trasfondo. Pero al final, tanto C# como Visual Basic deben seguir las reglas de .NET, y aunque nosotros como usuarios no tengamos que preocuparnos, los compiladores sí que lo harán.

Guillermo "Guille" Som es Microsoft MVP de Visual Basic desde 1997. Es redactor de dotNetManía, miembro de *Ineta Speakers Bureau Latin America*, mentor de *Solid Quality Learning* Iberoamérica y autor del libro *Manual Imprescindible de Visual Basic .NET*.
http://www.elguille.info

Independientemente de las bromas, tenemos que ser conscientes (sobre todo los programadores de Visual Basic) de que algunas veces el que nos "mimen" tanto no es bueno, ya que nos acostumbran mal, y cuando creemos que todo va a ser sencillo, llega la versión 2005 y nos dicen que si queremos usar la nueva instrucción `Custom Event` debemos saber manejar los delegados, además de que también debemos saber en qué medida están relacionados con los eventos. Esto a los programadores de C# no les pillaría tan desprevenidos. Así, si ahora les dicen que pueden crear métodos anónimos, y que esos métodos anónimos los podrán crear donde se pueda usar un delegado, o que ya no es necesario usar un constructor para crear un tipo delegado o que por medio de la *covarianza* o la *contravarianza* podrán usar de forma más óptima los delegados, simplemente estarán preparados y sabrán soportar el cambio...

Pero como siempre hay gente nueva, (tanto en C# como en Visual Basic), no está de más que algunos puntos estén totalmente claros, así que eso es lo que vamos a intentar en este primer artículo dedicado a los delegados y a los eventos. Y en los que seguirán, terminaremos por aclarar casi cualquier duda que posiblemente se nos pueda presentar a la hora de trabajar con los eventos y con los delegados.

¿Qué son y para qué sirven los delegados?

Como sabemos, .NET Framework se caracteriza por ser un entorno de código administrado (*managed code*) o lo que es lo mismo, a .NET no le gustan las sorpresas. Si una función tiene que devolver un valor de tipo `String`, debe devolver un valor de tipo `String`; si un método debe recibir dos parámetros de tipo `Object`, eso es lo que recibirá. Y todo esto los lenguajes adscritos a .NET deben respetarlo, ya que de no ser así, .NET no permitirá la ejecución del código. Y esto es aplicable a todo lo que .NET controla, es decir, a todo lo que está bajo su influencia. Lo que no quiere decir que no podamos hacer cosas que .NET no permita; pero si lo hacemos, debemos hacerlo por la puerta falsa. De esto saben mucho los que han desarrollado con C, incluso los que desarrollan con C#. Aunque en todas las puertas falsas de .NET siempre hay alguien que "está por allí" y revisa que en realidad no hagamos demasiadas trastadas. Esto en otros lenguajes no es así, y por error o porque así lo hayamos previsto, podemos crear grandes problemas, si no, ¿por qué aparecen los fallos de protección general? (las típicas pantallas azules de las versiones anteriores de Windows, que ahora simplemente están remozadas y han cambiado de *look* por un cuadro de diálogo más "mono").

arbitrario, es decir, que no nos "colemos" donde no debemos, ya que, como ya he dicho, a .NET no le gustan las sorpresas, por eso impone reglas que debemos cumplir; si las cumplimos, pasamos, si no las cumplimos, no nos deja seguir.

Y esto es así por todo lo comentado anteriormente, ya que el CLR quiere seguridad y la única forma de tenerla es creando normas de conducta y de utilización, en este caso, de la memoria o del acceso a esas partes de la memoria en la que están las definiciones de los métodos o funciones.

> Un delegado permite acceder a una función de forma casi anónima, ya que simplemente tiene la dirección de memoria de dicha función

Este tipo de problema se debe a un acceso indebido a la memoria, normalmente causado por un acceso a una posición de memoria que no estaba dentro del rango que teníamos permitido. .NET es más estricto y menos permisivo para estas cuestiones, por tanto, si queremos estar bajo el abrigo de la seguridad de .NET debemos seguir sus normas.

Como sabemos, .NET define una serie de tipos de datos, los cuales podemos usar indistintamente desde un lenguaje u otro, ya que independientemente del nombre que cada compilador le dé, en realidad estamos trabajando con los tipos definidos en la librería de clases, o mejor dicho, en el sistema de tipos comunes (CTS). Y los delegados no son una excepción.

Pero... ¿qué es un delegado? Un delegado es una referencia a una función, lo que también se conoce como un puntero a una función, es decir, un delegado permite acceder a una función de forma casi anónima, ya que simplemente tiene la dirección de memoria de dicha función. Y sabiendo la dirección de memoria, podemos acceder a ella. Pero en .NET esto debe estar controlado, de forma que ese acceso no sea

Por tanto, si queremos acceder a un método, tenemos que hacerlo por medio de un puntero controlado, y la forma de controlar ese acceso es definiendo un prototipo en el que indiquemos de qué tipo es ese método, si recibe parámetros, y de hacerlo cuántos y de qué tipo son. Una vez que tenemos definidos todos estos requerimientos, es cuando le decimos al *runtime* de .NET que nos permita acceder a esa función. De esta forma, podrá controlar que estamos accediendo al sitio correcto.

Esa definición del prototipo de función (o método) al que queremos acceder lo hacemos por medio de un delegado. Podemos pensar que un delegado es en cierto modo similar a las interfaces (ver **dotNetManía** nº 16), que como sabemos definen un contrato que debemos respetar. Sabiendo esto, si queremos acceder a una función que devuelve una cadena y que recibe un parámetro de tipo `Cliente`, debemos definir un delegado con esas características, y cuando posteriormente queramos acceder a ese método, en lugar de hacerlo directamente o por medio de un puntero directo, usaremos un objeto del tipo definido por el delegado. Lo que nos lleva a una segunda definición de lo

Los delegados definen la "firma" que los métodos a los que queremos acceder deben tener

que es un delegado, en la que podemos decir que es un tipo especial que nos permite definir la forma de acceder a una función.

Veamos el ejemplo del método que devuelve una cadena y recibe un parámetro de tipo **Cliente**. Ese método lo podemos definir en C# de esta forma:

```
public string MiFuncion(Cliente c)
                        {return "...";}
```

La forma de usar ese método sería algo así:

```
string s = MiFuncion( new Cliente() );
```

Por supuesto aquí no estamos usando ningún puntero, simplemente estamos accediendo a ese método de forma directa. Es más, debido a que no estamos indicando dónde está dicha función, suponemos que estamos accediendo desde la propia clase en la que está definido. Pero si quisiéramos acceder de una forma más anónima, podríamos definir un delegado que tenga esa definición y solo tendríamos que indicar dónde está definido para permitirnos el acceso, sin necesidad de usar una instancia de la clase que lo define.

¿Por qué tanta complicación? Porque debemos suponer que en ciertas circunstancias no tenemos una forma directa de acceder a ese método, ya que si la tuviéramos, no tendríamos necesidad de complicarnos la vida y usaríamos la forma directa mostrada anteriormente. Por ejemplo, si queremos que desde otra parte del código alguien pueda acceder a ese método sin necesidad de saber si está definido en una clase o en otra, podemos usar un delegado y por medio de dicho delegado acceder al método.

Veamos la definición del delegado que nos permitiría acceder a esa función y cómo podemos usarlo. Primero definimos el delegado usando la misma "firma" que tiene el método al que queremos acceder:

```
public delegate string MiFuncionDelegado(Cliente c);
```

Fuente 1. Definición de un delegado para acceder a un método definido con la misma firma.

Para usar el delegado, definimos una variable de ese tipo, y como los delegados en realidad son como clases, podemos usar el mismo código que usamos para crear cualquier tipo, con la diferencia de que en el constructor debemos indicarle la función a la que queremos acceder:

```
MiFuncionDelegado mfd = new
       MiFuncionDelegado(MiFuncion);
```

En este caso, también estamos accediendo desde la propia clase al método **MiFuncion**, pero lo dejamos así para mantener las cosas simples.

Para acceder a esa función por medio del delegado que acabamos de crear, lo haremos así:

```
string s = mfd( new Cliente() );
```

Que como podemos comprobar, es un código muy parecido al usado anteriormente, pero con la diferencia de que en este código no usamos el nombre de la función, sino el del objeto creado a partir del delegado.

Ahora supongamos que este último código lo queremos usar en cualquier método de cualquier clase, sin importar cómo y dónde se haya definido, ¿cómo podríamos hacerlo? porque para poder definir una variable como en este caso, deberíamos tener "cercana" la función a la que queremos acceder. Y si la tenemos cerca, pues no necesitamos usar un delegado. Veamos el siguiente código y seguro que esa cercanía no es tan necesaria:

```
public static void usarMiFuncionDelegado(
                    MiFuncionDelegado mfd)
{
   string s = mfd( new Cliente() );
}
```

En este caso, tenemos un método que define un parámetro del tipo del delegado, por tanto, podemos llamar a ese método pasándole la dirección de una función que cumpla con la definición del delegado y no importará donde estén definidos, ni el método ni la función a la que accederemos por medio del delegado. La única condición es que desde donde hagamos la llamada, tengamos acceso a ambos métodos (o funciones), pero no tienen porqué estar todas definidas en una misma clase, ¡ni siquiera en un mismo ensamblado!

Para acceder a este método lo podemos hacer de esta forma:

```
static void Main(string[] args)
{
   Cliente c = new Cliente();
   usarMiFuncionDelegado(c.MiFuncion);
}
```

Debemos notar en la forma en que llamamos al método que recibe el delegado, ya que simplemente le hemos pasado el nombre de la función. Esto es nuevo en C# 2.0 (aunque no en Visual Basic), y la forma en que tendríamos que hacerlo en las versiones anteriores sería esta otra:

```
usarMiFuncionDelegado(new
     MiFuncionDelegado(c.MiFuncion));
```

Es decir, pasando como argumento un objeto creado a partir del "tipo" del delegado.

De igual forma, el código que mostramos anteriormente en el que usábamos también un constructor de ese tipo lo podríamos haber escrito de esta otra forma, que como vemos, es más simple e igualmente comprensible.

```
MiFuncionDelegado mfd = MiFuncion;
```

Fuente 2. C# 2.0 permite asignar directamente la función sin necesidad de un constructor.

NOTA

En Visual Basic, la forma de acceder a una función siempre es por medio de la instrucción `AddressOf` y la podemos usar también de las dos formas que acabamos de ver, es decir por medio de un objeto del tipo del delegado o de forma directa, en cuyo caso, (al igual que ocurre con C# 2.0), será el propio compilador el que determinará si esa función cumple o no los requisitos del parámetro que espera el método:

```
usarMiFuncionDelegado(AddressOf c.MiFuncion)

usarMiFuncionDelegado(New
            MiFuncionDelegado(AddressOf c.MiFuncion))
```

En las asignaciones también podemos usar cualquiera de las dos formas:

```
Dim mfd As MiFuncionDelegado
mfd = New MiFuncionDelegado(AddressOf MiFuncion)

Dim mfd As MiFuncionDelegado = AddressOf MiFuncion
```

En cualquier caso, lo que estamos pasando es la dirección de memoria de la función.

De todo lo que hemos visto debemos concluir que los delegados definen la "firma" que los métodos a los que queremos acceder deben tener, y que podemos acceder a esos métodos por medio de instancias creadas a partir del delegado, y si estamos usando C# 2.0 el compilador se puede encargar de averiguar qué delegado es el que debe usarse y lo usará de forma transparente para nosotros.

Antes de ver qué relación tiene todo esto con los eventos, repasaremos algunas otras características de los delegados que en el fondo también están relacionadas con los eventos, o en la forma que finalmente las utilizan los eventos, pero que no necesariamente usaremos para trabajar con los eventos.

Usos prácticos de los delegados

Para que nos quede más claro el funcionamiento de los delegados, vamos a ver en qué situaciones podemos usarlos. Algunos de los usos que vamos a ver son exclusivos de C# 2.0, y aunque ya se han tratado en otros números de esta revista, no viene mal darles un repaso. Debido a esa exclusividad de uso en C#, decirle a los lectores que prefieren Visual Basic que no pasen al siguiente artículo, ya que aún quedan cosas que explicar que también son válidas para ese lenguaje, aunque (como es costumbre en esta sección), el código mostrado será prácticamente en exclusiva para C#. En otro artículo trataremos temas que son específicos de Visual Basic y por consiguiente todo el código será en ese lenguaje, ya que como comenté anteriormente los delegados son una pieza clave para los eventos, y debemos saber cómo funcionan los delegados para comprender mejor cómo funcionan los eventos.

Novedades de C# respecto a los delegados

En la versión 2.0 de C# (tal como explicó **Octavio Hernández** en el número 20 de esta revista) podemos usar los delegados de forma directa, es decir, sin necesidad de que tengamos que crear una instancia de la clase del delegado al que le pasamos como parámetro del constructor la función a la que queremos apuntar, tal como hemos visto en el código del fuente 2. En esos casos, el compilador comprueba el tipo de la variable que recibe el puntero a la función y si tiene la misma firma que la función, será el propio compilador el que haga el uso adecuado del delegado correspondiente. De hecho, si examinamos el código IL generado por el compilador será el mismo en los dos casos.

```
void usaMiFuncion2(){
    MiFuncionDelegado mfd = MiFuncion;
    string s = mfd(new Cliente());
}
void usaMiFuncion3(){
    MiFuncionDelegado mfd = new
          MiFuncionDelegado(MiFuncion);
    string s = mfd(new Cliente());
}
```

Fuente 3. Las dos formas equivalentes de asignar el puntero a una función.

```
ldftn  instance string Cliente::MiFuncion(class Cliente)
newobj instance void MiFuncionDelegado::.ctor(object,native int)
stloc.0
ldloc.0
newobj   instance void Cliente::.ctor()
callvirt instance string MiFuncionDelegado::Invoke(class Cliente)
stloc.1
ret
```

Fuente 4. El código IL es el mismo para las dos formas mostradas en el fuente 3.

En el código del fuente 3 tenemos las dos formas de realizar la asignación y en el fuente 4 vemos el código IL que crea el compilador, el cual es exactamente el mismo en ambos casos.

Métodos anónimos

Los métodos anónimos son otra de las novedades de C# que están relacionadas con los delegados. Y es que en C# 2.0 podemos usar una definición de un delegado en cualquier sitio que el compilador esperaría que se asignara un delegado.

El ejemplo más claro del uso de los métodos anónimos es para relacionar un evento con un código, pero como aún no hemos tratado con detalle los eventos, vamos a verlo con el código que estamos usando últimamente. En el código del fuente 5 podemos ver cómo crear un método anónimo con la misma firma que el delegado `MiFuncionDelegado`, pero al definirlo nosotros de forma independiente podemos escribir en el cuerpo de la función anónima lo que creamos conveniente.

```
public static void usaMiFuncion4()
{
  MiFuncionDelegado mfd;
  mfd = delegate(Cliente c)
        {return "Hola " + c.Nombre;};
  string s = mfd(new Cliente("Pepe"));
  Console.WriteLine(s);
}
```

Fuente 5. Definición de un método anónimo.

Como es evidente, en este caso no hace falta usar un método anónimo, ya que sería más fácil mostrar directamente el saludo, en lugar de dar tantas vueltas, pero lo importante es ver cómo se

pueden usar los métodos anónimos, los cuales tienen mayor utilidad con los eventos o cuando queramos simplemente definir la función "apuntada" de forma directa, ya que como hemos visto anteriormente, esa variable que apunta al método la podemos pasar como argumento a otro método.

En cualquier caso, lo que no nos estará permitido hacer es modificar la firma del delegado; por tanto, si se nos ocurre la brillante idea de añadir un nuevo parámetro a la función, tal como vemos en el siguiente código fuente, el compilador nos avisará (entre otras cosas), que `MiFuncionDelegado` no tiene dos argumentos.

```
mfd = delegate(Cliente c, string saludo){
    return saludo + " " + c.Nombre;
};
```

Es importante saber que aunque estemos declarando un "método" anónimo, en realidad no es un método, al menos en el sentido de los ámbitos o cobertura de las variables, ya que desde ese método anónimo podemos acceder a cualquier variable que hayamos declarado anteriormente, y la variable definida en el parámetro también tendrá el mismo ámbito que el método o propiedad que contiene esa definición anónima. Debemos pensar en que el cuerpo del método anónimo en realidad es como cualquier otro bloque de código que podamos incluir dentro de un par de llaves.

En el código del fuente 6 podemos ver ese conflicto entre la variable definida directamente y la definida como parámetro del método anónimo.

```
Cliente c = new Cliente();
usarMiFuncionDelegado(c.MiFuncion);

MiFuncionDelegado mfd;
mfd = delegate(Cliente c) {return "Hola " + c.Nombre;};
string s = mfd(new Cliente("Manolo"));
Console.WriteLine(s);
```

Fuente 6. El ámbito de un método anónimo es el mismo que el del bloque en el que se define.

Covarianza y contravarianza

La primera vez que leí estas dos palabras, pensé que el tema tratado debía ser muy complicado, seguramente para gente más experta que yo en C#. Y como soy un catetillo de pueblo que no tiene estudios, después de leer la descripción de la ayuda de Visual Studio 2005 pensé en la suerte que tenía de que mi lenguaje materno no fuese el C#. Pero si iba a hablar de esto en este artículo, lo lógico era que me empapara del tema, al menos si lo iba a tratar. Y con esto pasa como con casi todo, hasta que no lo tienes entre las manos, no sabes el tacto que tiene.

Sobre la *covarianza*, la documentación nos dice que: *Cuando un método delegado tiene un tipo de valor devuelto que es más derivado que la firma de delegado, se denomina covariante*. En realidad, al leerlo sabiendo qué es lo que significa no era tan rebuscada la definición. Pero para dejarlo en un lenguaje más llano, diremos que esto significa que podemos usar delegados que devuelvan un tipo y el receptor de ese valor puede ser cualquier clase de ese mismo tipo o de cualquier otra clase derivada.

Por ejemplo, en el siguiente código, tenemos la definición de una clase `Persona` y un delegado que devuelve un valor de ese tipo:

```
public delegate Persona PersonaCallback();

public class Persona
{
   // Omitidas las definiciones
   // de las propiedades Nombre y Apellidos
}
```

Y definimos una clase derivada, a la que llamaremos `Colega`, en la que definimos un método estático que devuelve un objeto de ese mismo tipo:

```
public class Colega : Persona
{
    public static Colega NuevoColega()
    {
        return new Colega();
    }
    // Omitida la definición de la propiedad Correo
}
```

Podemos usar esa función como el método al que apuntará el delegado `PersonaCallback`, que como hemos visto, devuelve un objeto del tipo `Persona`:

```
PersonaCallback nColega;
nColega = Colega.NuevoColega;

Colega unColega = (Colega)nColega();
// Omitidas las asignaciones a las propiedades
```

En la asignación a la variable `unColega` debemos hacer una conversión (*cast*) ya que el valor devuelto por el delegado es del tipo `Persona`, independientemente del tipo que devuelva en realidad la función a la que hace referencia ese delegado.

En las versiones anteriores, para conseguir esto mismo teníamos que definir un delegado para cada una de los tipos que quisiéramos devolver. Ni que decir tiene que esto solo lo podemos hacer con C#; en Visual Basic no está permitida esta forma de usar los valores devueltos por un delegado.

Veamos ahora qué nos dice la documentación sobre la *contravarianza*: *Cuando una firma de método delegado tiene uno o más parámetros de tipos que derivan de los tipos de los parámetros de método, ese método se denomina contravariante*. Lo que viene a significar que los tipos de

datos que podemos usar como parámetros al llamar a un delegado pueden ser del mismo tipo que está definido en el delegado (así era hasta la versión 2.0), o de cualquier tipo derivado. Si el compilador ve una relación de herencia entre el tipo usado y el definido en el delegado, lo permitirá.

Siguiendo con el ejemplo del delegado que recibe un parámetro de tipo **Cliente**, (ver el fuente 1), vamos a rediseñar la clase y el método **MiFuncion** para que devuelva un saludo al nombre indicado en la propiedad **Nombre** de la clase. A continuación creamos la clase **ClienteOro** que se deriva de **Cliente**. Tanto en una como en otra clase hemos definido un constructor que recibe como parámetro el nombre a usar. En el fuente 7 vemos esas dos definiciones de estas clases.

```
public class Cliente
{
    // Omitida la definición de la propiedad Nombre
    public Cliente() { }
    public Cliente(string nombre)
    { this.Nombre = nombre; }
    //
    public virtual string MiFuncion(Cliente c) {
        return "Que tal " + c.Nombre;
    }
}

class ClienteOro : Cliente
{
    public ClienteOro() { }
    public ClienteOro(string nombre)
    { this.Nombre = nombre; }
}
```

Fuente 7. Definición simplificada de las clases **Cliente** y **ClienteOro**.

NOTA

El hecho de definir un constructor con parámetro en las clases del fuente 7 es para facilitar el uso de las mismas, de forma que en una sola instrucción podamos asignar el valor de la propiedad **Nombre**, de esa forma nuestro código de ejemplo podrá mostrar algo. Pero la razón de que lo hayamos tenido que hacer en las dos clases es porque los constructores no se heredan; por tanto, si queremos esa funcionalidad en las dos clases, debemos definirlos en ambas.

Ahora podemos usar cualquier función que reciba un delegado que apunte a **MiFuncion** y el compilador (realmente el *runtime*) usará la clase que corresponda.

```
ClienteOro co = new ClienteOro("Paco");
MiFuncionDelegado mfd2;
mfd2 = co.MiFuncion;
string sco = mfd2(co);
Console.WriteLine(sco);
```

Incluso podemos crear un método anónimo que use un objeto del tipo **ClienteOro** como parámetro, pero siempre y cuando el método anónimo tenga la firma correcta, es decir, el parámetro en la definición del método anónimo debe ser del mismo tipo que el indicado en la definición del delegado, pero a la hora de usarlo podemos indicar cualquier objeto de un tipo **Cliente** o derivado:

```
MiFuncionDelegado mfd3;
mfd3 = delegate(Cliente co2) {return "A sus pies " +
                                co2.Nombre;};
Console.WriteLine(mfd3(co));
```

Pero el uso más práctico de esta característica será (como casi todo lo relacionado con los delegados), cuando lo apliquemos a los eventos.

Cuando utilizamos los métodos de eventos de los controles de Windows Forms, el segundo parámetro suele (o debería) ser una clase derivada de **EventArgs**, pero dependiendo del evento, ese parámetro será del tipo adecuado para el evento en cuestión. Por ejemplo, el evento **KeyPress** recibe un parámetro del tipo **KeyPressEventArgs**, pero en C# podemos definir el método que intercepta ese evento de cualquiera de estas dos formas:

```
private void txtNombre_KeyPress( object sender,
                                 KeyPressEventArgs e)
   { ... }

private void txtNombre_KeyPress( object sender,
                                 EventArgs e)
   { ... }
```

La segunda forma, a pesar de ser menos específica, seguramente la usaremos en casos muy concretos y siempre que necesitemos esa "generalidad", pero no adelantemos acontecimientos, ya que cuando tratemos el tema de los eventos veremos algunas aplicaciones prácticas de esta posibilidad que tiene C#, ya que en Visual Basic siempre tendremos que definir los parámetros de los eventos del tipo exacto.

Conclusiones

Aún no hemos terminado con algunas de las cosas interesantes o importantes de los delegados, pero será en el próximo artículo donde veremos otra característica interesante de los delegados: la multidifusión. Esa forma de usar los delegados la comprenderemos mejor cuando sepamos más sobre la estrecha relación de estas clases especiales con los eventos.

Como es costumbre en esta sección, en el sitio Web de **dotNetManía** está disponible el código de ejemplo para poder bajarlo, aunque en el caso de Visual Basic no será equivalente al de C#, simplemente porque aquél no soporta algunas de las características de los delegados que hemos tratado.