Eventos “ala” .NET

Recientemente ha surgido en los grupos de C# una duda sobre si los eventos que usa el .NET son del mismo tipo que los que nosotros podemos crear y usar. La respuesta obvia y evidente es que sí, son exactamente lo mismo y funcionan de la misma forma. Aprovechando que el otro día encontré algo sobre esto en el libro de ***C++/CLI in Action*** de Nishtant Sivakumar voy a parafrasear el contenido aquí con bastante adiciones propias.

Básicamente un evento es un manejador que contiene un delegado y que se suele utilizar para que algún objeto notifique a otro de que se producido un hecho. Es una visión ampliada de la función *callback* típica de Win32.

Dentro del .NET los eventos se utilizan para notificar a nuestros objetos que se ha producido algún tipo de hecho al que nos hemos suscrito.

Imaginemos que queremos que el .NET nos notifique cada vez que el usuario pulse una tecla en un campo de edición. La solución políticamente oficial es la de crear un método de nuestra ficha y asociar dicho método al evento de la siguiente manera:

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{

}
```

El bloque de código de arriba representa el método que se invocará cuando se produzca el evento de la pulsación de la tecla. Es un método normal y corriente perteneciente a una ficha pero que presenta una firma concreta: recibe un parámetro de tipo *Object* y otro que es un objeto de una clase llamada *KeyPressEventArgs*.

Pero el editor del Visual Studio ha hecho más trabajo por nosotros:

```
this.textBox1.KeyPress += new
       System.Windows.Forms.KeyPressEventHandler(this.textBox1_KeyPress);
```

Dentro del método *InitializeComponent()* ha añadido la línea de arriba. Lo que está haciendo es asociar nuestro método con el evento *KeyPress* del objeto que representa nuestra caja de edición. *KeyPress* es un evento, que se asocia a un delegado del tipo *KeyPressEventHandler*, que realmente es nuestro método, es decir, la parte “*new …”* lo que hace es añadir nuestro método a la lista de eventos *KeyPress* de la instancia del componente *TextBox*. Personalmente pienso que la notación es bastante desafortunada, ya que realmente estamos llamando a una serie de métodos ocultos que el compilador sustituye por nuestro código.

Para decirlo fácil: *KeyPress* es una bolsa que contiene una lista de delegados que se llamarán en el mismo orden en que se introdujeron, y nosotros estamos añadiendo a esa bolsa nuestro propio delegado para que se llame cuando sea necesario.

Luego, internamente, y a nivel del sistema operativo y de Win32, dicho objeto comparte entre todos los objetos del mismo tipo existentes en la aplicación, una función llamada **función del bucle de mensajes** que Windows ejecuta cada vez que se produce algo en un objeto de ese tipo. Dicha

función recibe lo que se llama **mensajes de Windows**, que no es otra cosa que una serie de variables del tipo *unsigned long* y que contienen datos sobre la situación producida.

Esta función ha de atender a, por ejemplo, al mensajes WM_CHAR, que es el mensaje que Windows envía a dicha función cuando el usuario ha pulsado una tecla. La atención a dicho mensaje consiste en averiguar qué objeto de todos los existentes es el que ha recibido la pulsación **y en ejecutar o lanzar el evento correspondiente**, en nuestro caso y para .NET, invocar la lista de delegados almacenada en el *KeyPress* de nuestro *TextBox*.

NOTA: Al comenzar a explicar esto he dicho *la solución políticamente oficial* con toda intención, ya que de hecho no estamos obligados a utilizar los eventos de esta forma. Podemos disponer de eventos para todo lo que queramos, ya sea para componentes visuales o no.

Ahora el ejemplo del libro (en C#)

Una vez que hemos visto cómo el .NET trabaja con los eventos que representan mensajes de Windows, vamos a parafrasear lo que viene en el libro citado, pasándolo de C++/CLI a C#.

En primer lugar creamos la clase que va a servir de parámetro al evento al estilo .NET:

```
public class FindWordEventArgs: EventArgs
{
    private string m_word;
    private int m_index;
    public string Word
    {
        get { return m_word; }
        set { m_word = value; }
    }
    public int Index
    {
        get { return m_index; }
        set { m_index = value; }
    }
    public FindWordEventArgs(string word,int index)
    {
        Word=word;
        Index=index;
    }
}
```

Observamos que nuestra clase hereda de *EventArgs*, que es la clase base para el paso de argumentos a los eventos del .NET. En la documentación se puede ver una rica y compleja jerarquía de clases para representar (y encapsular) la mayoría de los mensajes de Windows, introduciendo en dicha clase los elementos necesarios para que podamos reaccionar a dicho mensaje.

Después definimos la firma de nuestro delegado:

```
public delegate void FindWordHandler(FindWordEventArgs args);
```

Y ahora la propia clase que contiene el evento:

```
public class WordFinder
{
    private string str;
    public WordFinder(string s)
    {
        str=s;
```

```
        }

        public void Find(string word)
        {
            int index = 0;
            foreach(string w in str.Split(' '))
            {
                if (word.CompareTo(w) == 0)
                    OnFindWord(new FindWordEventArgs(w, index));
                index++;
            }
        }
        public event FindWordHandler OnFindWord;
    }
```

Lo interesante está en el método *Find*. Observamos que cuando encontramos una coincidencia, realizamos la llamada al evento *OnFindWord*, al que le pasamos una nueva instancia de la clase que encapsula los datos del mensaje. En este caso, *FindWordEventArgs* podría representar al *KeyPressEventArgs* del mensaje de Windows, ya que contiene lo que nos interesa: la información que se nos pasa en el evento para que nuestro método pueda reaccionar.

La última línea es mágica:

```
        public event FindWordHandler OnFindWord;
```

Esta línea lleva dentro de sí una buena cantidad de código oculto; de hecho se encarga de realizar las siguientes acciones:

- Añadir un nuevo delegado a la lista.
- Quitar un delegado existente de la lista.
- Invocar todos los delegados de la lista, si los hubiera.
- Crear y borrar todo el espacio necesario para las operaciones de arriba.

Como siempre, dentro del .NET hay mucha magia que a los programadores de C++ no nos gusta, pero que a veces nos resulta útil.

Ahora vamos a crear una clase que va a utilizar una instancia de *WordFinder* (igual que antes usábamos una de un *TextBox*) y que va a instalar un delegado en la lista de eventos del evento *OnFindWord*:

```
    public class ProgrammingLanguages
    {
        public void WordFoundHandler(FindWordEventArgs args)
        {
            Console.WriteLine("Word Found: {0} in position {1}", args.Word,args.Index);
        }
        public void SearchWord(string word)
        {
            WordFinder wf = new WordFinder("C++ VB Java Cobol C++ Java Fortran C++ Lisp");
            wf.OnFindWord += new FindWordHandler(WordFoundHandler);
            wf.Find(word);
        }
    }
```

Las cosas están claras dentro del método *SearchWord*: Creamos un objeto del tipo *WordFinder* (como el IDE del Visual Studio nos crea una instancia al *TextBox* dentro de *InitializeComponents*), añadimos nuestro método que será invocado como un evento por la clase y que nos notificará que se ha encontrado una palabra, y finalmente buscamos la palabra **C++.**

Cada vez que aparezca dicha palabra en la cadena, se disparará el evento. Y es así de fácil.

Código oculto (otra vuelta de tuerca)

Pero vamos a avanzar un poco más. Vamos a crear un evento… que sólo permita almacenar un delegado, y de paso vamos a mostrar algo de ese código oculto que el compilador de C# añade para nosotros.

Lo primero a añadir es la línea

```
private FindWordHandler _FindWordHandler;
```

que nos va a servir para almacenar el delegado que invocará el evento (una de las cosas que si no decimos nada, hace el propio C#).

Luego, cambiamos la línea

```
public event FindWordHandler OnFindWord;
```

por

```
public event FindWordHandler OnFindWord
{
    add
    {
        if(_FindWordHandler==null)
            _FindWordHandler=value;
    }
    remove
    {
        _FindWordHandler = null;
    }
}
```

De esta forma estamos definiendo (y sobrescribiendo) el comportamiento por defecto. Si nos damos cuenta la sintaxis es similar a la de las propiedades, pero con otras palabras reservadas.

Con *add* definimos qué ocurre cuando se añade un delegado al evento: en nuestro caso, si el delegado de almacén está vacío, asignamos el evento y no hacemos nada si ya está asignado, pero podríamos lanzar una excepción o hacer cualquier otra cosa.

Con *remove* eliminamos el delegado, pero seguimos teniendo libertad para hacer lo que nos de la gana ahí dentro.

El código por defecto, según la documentación, debería ser:

```
public event FindWordHandler OnFindWord
```

```
        {
            add
            {
                _FindWordHandler+=value;
            }
            remove
            {
                _FindWordHandler -= value;
            }
        }
```

Que lo que hace es añadir y eliminar el delegado respectivamente.

Y ahora, sin manos (es decir, en C++/CLI)

Voy ahora a repasar de forma rápida el ejemplo del libro, centrándome sólo en un par de detalles. Primero el código completo:

```
#include "stdafx.h"

using namespace System;

ref class FindWordEventArgs:EventArgs
{
public:
	property String ^Word;
	property int Index;
	FindWordEventArgs(String ^word,int index)
	{
		Word=word;
		Index=index;
	}
};

delegate void FindWordHandler(FindWordEventArgs ^);
ref class WordFinder
{
	String ^str;
	FindWordHandler ^_FindWordHandler;
public:
	WordFinder(String ^s):str(s),_FindWordHandler(nullptr){}
	void Find(String ^word)
	{
		int index=0;
		for each(String ^w in str->Split(' '))
		{
			if(word->CompareTo(w)==0)
				OnFindWord(gcnew FindWordEventArgs(w,index));
			index++;
		}
	}
	event FindWordHandler ^OnFindWord
	{
		void add(FindWordHandler ^handler)
		{
			if(_FindWordHandler==nullptr)
				_FindWordHandler=handler;
		}
		void remove(FindWordHandler ^)
		{
			_FindWordHandler=nullptr;
		}
		void raise(FindWordEventArgs ^args)
```

```
                {
                        if(_FindWordHandler)
                                _FindWordHandler(args);
                }
        }
};

ref class ProgrammingLanguages
{
        void WordFoundHandler(FindWordEventArgs ^args)
        {
                Console::WriteLine("Word Found: {0} in position {1}"
                            , args->Word,args->Index);
        }
public:
        void SearchWord(String ^word)
        {
                WordFinder ^wf = gcnew WordFinder(
                        "C++ VB Java Cobol C++ Java Fortran C++ Lisp");
                wf->OnFindWord +=
                        gcnew FindWordHandler(this,&ProgrammingLanguages::WordFoundHandler);
                wf->Find(word);
        }
};
int main(array<System::String ^> ^args)
{
        ProgrammingLanguages l;
        l.SearchWord("C++");
        return 0;
}
```

Aparte de las propiedades por defecto en la definición de la clase *FindWordEventArgs*, otra cosa interesante es que también podemos implementar el método *raise* del evento, pudiendo afinar mucho más nuestro evento.

Y eso es todo.