Guillermo "Guille" Som

Parámetros personalizados en los eventos

En este cuarto artículo dedicado a los delegados y eventos nos centraremos en cómo comunicarnos entre la clase que define los eventos y la que los intercepta. Veremos esa comunicación de dos formas diferentes, usando parámetros por referencia y de la forma recomendada, que es definiendo nuestra propia clase para usar como parámetro de los eventos. También abordaremos una característica exclusiva de C#, que es la posibilidad de usar clases base como parámetros de los métodos que reciben los eventos.

Comunicarse con la clase que produce el evento

Hasta ahora hemos estado usando los eventos de la forma habitual: para comunicarle a la aplicación receptora que algo ha ocurrido, y en esa comunicación solo hemos usado una vía, de forma que la clase que produce el evento "informe" de qué es lo que está ocurriendo, información que enviamos con los parámetros pasados al método receptor. Pero, ¿cómo podemos hacer que la aplicación receptora (el cliente) pueda comunicarse con la clase que produce el evento (el servidor), por ejemplo, para cancelar una acción o para pasarle cierta información?

Usando parámetros por referencia en los eventos

En principio, esto es algo sencillo de realizar, ya que lo único que tenemos que hacer es declarar por referencia uno (o varios) de los parámetros. De esta forma podemos asignar un valor a ese parámetro y usarlo en la clase que produce el evento. Por ejemplo, si tenemos el evento usado en los ejemplos de los artículos anteriores que avisa del cambio en el valor de una propiedad, podemos agregar un tercer parámetro para indicar si cancelamos dicha asignación. En el código del fuente 1 vemos cómo definir el delegado, el evento y la propiedad, que tiene en cuenta esta nueva posibilidad de cancelar la asignación, dato que sabremos si la aplicación cliente devuelve un valor verdadero en la variable usada para el tercer parámetro.

Como es de esperar, el método de la clase cliente que recibe la notificación de que el evento se ha producido debe tener la misma "firma" que el delegado, incluyendo el parámetro por referencia y, como es natural, podemos ignorar dicho parámetro o asignarle un valor verdadero en el caso de que queramos cancelar la asignación. En el código del fuente 2 cancelamos esa asignación si el nuevo valor es una cadena vacía o la longitud de la misma es inferior a 3 caracteres.

```
public delegate void NombreCambiadoEventHandler(
  string nuevo, string anterior, ref bool cancelar);

public event NombreCambiadoEventHandler NombreCambiadoRef;

private string m_Nombre;
public string Nombre
{
  get { return m_Nombre; }
  set
  {
    // Lanzar el evento
    // indicando el nuevo valor y el anterior,
    // además de la variable para saber si cancela
    if( NombreCambiadoRef != null )
    {
      bool cancelado = false;
      NombreCambiadoRef(value, m_Nombre, ref cancelado);
      if( cancelado )
      {
          // Lo que haya que hacer para avisar
          // que se ha cancelado la asignación
          return;
      }
    }
    m_Nombre = value;
  }
}
```

Fuente 1. Definición de un evento que permite cancelar la acción

Guillermo "Guille" Som
Es Microsoft MVP de Visual Basic desde 1997. Es redactor de **dotNetManía**, mentor de Solid Quality Iberoamericana, tutor de campusMVP, miembro de Ineta Speakers Bureau Latin America, y autor de los libros "Manual Imprescindible de Visual Basic .NET" y "Visual Basic 2005".
http://www.elguille.info

```
static void cli_NombreCambiadoRef(
  string nuevo, string anterior, ref bool cancelar)
{
  // Cancelamos la acción si el nuevo valor no
  // cumple las condiciones que estimemos oportunas
  if( string.IsNullOrEmpty(nuevo) || nuevo.Length < 3 )
  {
    Console.WriteLine( "cancelando... debe tener más" +
                       " de 2 caracteres...");
    cancelar = true;
  }
}
```

Fuente 2. El método que recibe el evento puede cancelar la asignación

Hay que aclarar que esa comprobación que hacemos en el lado del cliente en realidad la podíamos hacer en la propiedad, pero de lo que aquí se trata es de ver un ejemplo de cómo cancelar (o informar a la clase que ésa es nuestra intención), no de la solución mágica a todos nuestros problemas. Además, al hacerlo en el lado del cliente, esto nos da la posibilidad de que cada aplicación que use esa clase decida la comprobación que debe hacer. Por ejemplo, si la propiedad fuera una contraseña, el cliente podría validar si cumple ciertos requisitos impuestos por la aplicación.

Usando clases como parámetro de los eventos

Si necesitamos pasar más datos (ya sean por valor o por referencia), tendremos que indicar más parámetros en la definición del delegado asociado con el evento, y aunque eso no es ningún problema, .NET nos permite una forma más elegante de hacerlo: usando un tipo personalizado como parámetro, es decir, una clase. Es más, el propio .NET Framework tiene una clase que podemos usar para estos menesteres: `EventArgs`. De hecho, todos los eventos de los controles del espacio de nombres `Windows.Forms` se basan (o se derivan) de esa clase; por tanto, podemos hacer lo mismo que hace el entorno en el que estamos programando, y crear nuestra propia clase basada en `EventArgs`. De esa forma también nos resultará más cómodo pasar (y recuperar) datos en ambos sentidos, con lo que de camino nos ahorramos la definición de parámetros por referencia, que entre otras cosas, no son deseables, al menos a la hora de dar claridad a nuestro código.

Crear clases para usar como parámetro de un evento

Para mantener las cosas sencillas, vamos a crear una clase que usaremos como parámetro de los eventos. Esta clase tendrá la misma funcionalidad que la mostrada en el código del fuente 1; es decir, la clase tendrá tres propiedades, dos de solo lectura y la tercera de lectura/escritura, ya que será la usada para saber si se quiere cancelar la acción de asignar un nuevo valor a la propiedad. Las otras dos, las que indican los valores nuevo y anterior, serán de solo lectura, ya que lo único que le interesa al método receptor del evento es saber qué valores contienen, y como son valores directamente relacionados con una propiedad de una clase, no deberíamos manipularlos, ya que si lo hacemos nos estaríamos saltando a la torera uno de los pilares de la programación orientada a objetos: la encapsulación. Pero... como somos los diseñadores de la clase, podemos hacer lo que mejor nos parezca, aunque no voy a ser yo el que incite a esa decisión.

Las dos propiedades que vamos a exponer como de solo lectura las vamos a definir con los dos bloques de código que habitualmente tienen todas las propiedades, que como sabemos son el bloque `get`, que nos permite obtener el valor de la propiedad, y el bloque `set`, que es el usado cuando asignamos el valor. Para dar esa funcionalidad de solo lectura a pesar de definir el bloque `set`, vamos a usar la nueva característica de .NET Framework 2.0 que nos permite cambiar el ámbito (o visibilidad) de uno de los bloques de las propiedades (ver el número 20 de **dotNetManía**). De esta forma, la clase expondrá públicamente solo el bloque `get`, y el bloque `set` lo dejaremos como `internal` (`Friend` en Visual Basic) para que solo podamos usarlo desde el propio ensamblado.

En el código del fuente 3 vemos la definición de la clase `DatosCambiadosEventArgs`, que es la que usaremos como parámetro del delegado y el evento definidos en la clase `Cliente` (ver el fuente 4). Ambas clases están definidas en una biblioteca de clases, que tendremos que añadir a las referencias del pro-

```
public class DatosCambiadosEventArgs : EventArgs
{
    // Propiedad de solo lectura cuando se accede
    // desde fuera del propio ensamblado
    private string m_Nuevo;
    public string Nuevo
    {
        get { return m_Nuevo; }
        internal set { m_Nuevo = value; }
    }

    // Propiedad de solo lectura cuando se accede
    // desde fuera del propio ensamblado
    private string m_Anterior;
    public string Anterior
    {
        get { return m_Anterior; }
        internal set { m_Anterior = value; }
    }

    private bool m_Cancelar;
    public bool Cancelar
    {
        get { return m_Cancelar; }
        set { m_Cancelar = value; }
    }
}
```

Fuente 3. La definición de la clase basada en `EventArgs`

```
public class Cliente
{
  public delegate void DatosCambiadosEventHandler(
          DatosCambiadosEventArgs e);

  public event DatosCambiadosEventHandler NombreCambiado;

  private string m_Nombre;
  public string Nombre
  {
    get { return m_Nombre; }
    set
    {
      // Lanzar el evento
      // indicando el nuevo valor y el anterior
      if( NombreCambiado != null )
      {
        // Declaramos un objeto del tipo de la clase
        // usada para usar como parámetro del evento.
        DatosCambiadosEventArgs e =
                new DatosCambiadosEventArgs();
        e.Cancelar = false;
        // Desde el propio ensamblado podemos acceder
        // al bloque set de las propiedadades Nuevo
        // y Anterior
        e.Nuevo = value;
        e.Anterior = m_Nombre;
        // Lanzamos el evento usando como argumento el
        // objeto creado a partir de la clase.
        NombreCambiado(e);
        // Comprobar si se cancela la asignación
        if( e.Cancelar )
        {
            // Lo que haya que hacer para avisar
            // que se ha cancelado la asignación
            return;
        }
      }
      m_Nombre = value;
    }
  }
  // No mostrado el resto de la definición de la
  // clase Cliente
```

Fuente 4. Parte de la clase `Cliente` en la que se definen los eventos que usan la clase definida en el fuente 3

```
static void cli_NombreCambiado(DatosCambiadosEventArgs e)
{
  if( string.IsNullOrEmpty(e.Nuevo) || e.Nuevo.Length <
3 )
  {
    Console.WriteLine("  cancelando... debe tener" +
                      " más de 2 caracteres...");
    e.Cancelar = true;
    // Si la clase que define el parámetro
    // está en otro ensamblado
    // esta propiedad será de solo lectura
    //e.Nuevo = "Miguel";
  }
}
```

Fuente 5. El método que intercepta el evento definido en el fuente 4

yecto en el que definimos la clase que interceptará los eventos y añadir la importación correspondiente del espacio de nombres que las contiene. Una vez hecho eso, la podremos usar tal como vemos en el código fuente 5.

Definir la clase del evento en el mismo ensamblado que el cliente

Si en lugar de crear un ensamblado para las clases (la que produce el evento y la que define el parámetro de dicho evento) decidimos que estén todas en el mismo ensamblado (proyecto), el hecho de proporcionar ámbitos diferentes a los bloques `set` de las dos propiedades que queremos que sean de solo lectura no nos soluciona la papeleta, ya que al estar declarados esos bloques con `internal`, serán accesibles desde cualquier parte del mismo ensamblado, y por tanto también desde el código de la aplicación cliente. Por lo tanto, en estos casos es preferible optar por la forma "recomendada" de definir esas propiedades como de solo lectura, y para poder asignarles el valor que tendrán lo haremos por medio de un constructor parametrizado. De esta forma, desde el constructor podremos asignar los valores correspondientes a los campos privados que siempre definimos para cada propiedad. En el código del fuente 6 tenemos la definición de la clase usada como parámetro de los eventos.

En el código del fuente 6 también podríamos haber usado la característica de tener diferentes niveles de accesibilidad en las propiedades, pudiendo definir

```
class DatosCambiadosEventArgs : EventArgs
{
    public DatosCambiadosEventArgs(
        string nuevo, string anterior)
    {
        m_Nuevo = nuevo;
        m_Anterior = anterior;
    }

    private string m_Nuevo;
    public string Nuevo
    {
        get { return m_Nuevo; }
    }

    private string m_Anterior;
    public string Anterior
    {
        get { return m_Anterior; }
    }

    private bool m_Cancelar;
    public bool Cancelar
    {
        get { return m_Cancelar; }
        set { m_Cancelar = value; }
    }
}
```

Fuente 6. Definición de la clase basada en `EventArgs` usando un constructor con parámetros

como `private` el bloque `set`, ya que solo necesitaríamos accederlo desde la propia clase que lo define, impidiendo de esa forma que se pueda asignar un valor desde fuera de la clase.

Métodos de apoyo para producir los eventos

Otra de las recomendaciones que atañen a los eventos que tenemos definidos en una clase (o tipo), es la de crear un método (habitualmente definido como protegido) que sea el que en realidad se encargue de producir el evento. De esa forma, cuando queramos lanzar un evento, en lugar de hacerlo directamente usaremos ese método, al que le pasaremos los parámetros correspondientes y será el encargado de producir el evento. La razón de hacerlo así, y la de definir esos métodos como `protected` (protegidos) es para que desde las clases derivadas tengamos una forma fácil de producir los eventos que define la clase base sin necesidad de escribir todo el código que necesitemos cada vez que tengamos que producir un evento. La nomenclatura recomendada para este tipo de métodos es que tenga el mismo nombre que el evento, pero empezando con `On`; en el caso del evento `NombreCambiado`, ese método se llamará `OnNombreCambiado`. Esos métodos de apoyo también suelen ser de tipo `void` (no devuelven un valor), pero en el ejemplo que estamos usando a lo largo de este artículo podemos hacer que devuelva un valor de tipo `bool`, de forma que nos indique si se cancela o no la asignación de la propiedad. El hacerlo así es porque es preferible que devuelva un valor verdadero o falso en lugar de usar un parámetro por referencia.

Para que tengamos las cosas claras, en el código del fuente 7 podemos ver la definición del método de apoyo para el evento `NombreCambiado`, además de ver cómo podemos usar ese método desde el bloque `set` de la propiedad `Nombre`.

Recordemos que el método de apoyo `OnNombreCambiado` devuelve un valor de tipo `bool` porque eso es lo que necesitamos, ya que desde la propiedad tenemos que saber si se cancela o no la asignación del nuevo valor. En el código completo que acompaña al artículo, la propiedad `Apellidos` también utiliza un método para lanzar el evento, pero en ese caso el valor de la propiedad `Cancelar` se ignora; por tanto, el método está declarado como `void` (`Sub` en Visual Basic), ya que no necesita devolver nada.

Eventos definidos en clases para usarlas en formularios

Todos los eventos de los controles de `Windows.Forms` siempre tienen dos parámetros: el primero hace referencia al objeto que produce el evento (el control) y el segundo contiene la información sobre el evento (normalmente una clase derivada de `EventArgs`). Si no ha de proporcionar ninguna información, este último

```
public string Nombre
{
  get { return m_Nombre; }
  set
  {
    // Lanzar el evento
    // indicando el nuevo valor y el anterior.
    if( NombreCambiado != null )
    {
      // Usamos el método de apoyo para lanzar el evento
      if( OnNombreCambiado(value, m_Nombre) )
      {
        // Lo que haya que hacer para avisar
        // que se ha cancelado la asignación
        Console.WriteLine( "  Cancelado el nuevo " +
                           "nombre: {0}.", value);
        return;
      }
    }
    m_Nombre = value;
  }
}

// Si vamos a usar un método de apoyo para lanzar
// el evento y vamos a tener en cuenta la propiedad
// Cancelar el método debería devolver el valor de
// cancelación
protected bool OnNombreCambiado(string nuevo, string
anterior)
{
  DatosCambiadosEventArgs e =
      new DatosCambiadosEventArgs(nuevo, anterior);
  e.Cancelar = false;
  NombreCambiado(e);
  return e.Cancelar;
}
```

Fuente 7. Definición y uso de un método de apoyo desde el que lanzamos el evento

parámetro se define del tipo base `EventArgs`, pero siempre se utiliza, más que nada para dar "consistencia" a la forma de declarar los eventos.

Sabiendo esto, debemos hacer un par de aclaraciones. La primera es que si nosotros definimos una clase basada en cualquier control, con idea de poder usarla en los formularios, y añadimos nuevos eventos, e incluso creamos nuestra propia clase para definir los parámetros de esos eventos, a la hora de producir el evento debemos tener en cuenta que el primer parámetro del delegado asociado a ese evento (y por tanto del método que recibe la notificación) es de tipo `object`, y representa al control que produce el evento. Como somos nosotros los que hemos definido esa clase, y desde ella lanzamos el evento, ese primer parámetro debe hacer referencia a la propia clase que produce el evento. Por otra parte, el segundo parámetro es de la clase que usamos para dar información al receptor del evento sobre lo que debe tener en cuenta en relación con ese evento que acaba de producirse; y aquí es cuando entra la segunda aclaración, ya que para ser "consistentes" con la forma de definir y usar los eventos en los formularios, nuestros eventos también deben tener siempre un segundo parámetro

derivado del tipo `EventArgs`; si no es necesario proporcionar información extra, al menos debemos usar la propia clase base.

En el código del fuente 8 vemos la definición del método asociado al evento de una clase derivada de `TextBox` que produce el evento `TextoCambiado` (el equivalente a `DatosCambiados` que hemos estado viendo hasta ahora).

```
// El delegado para el evento TextoCambiado
public delegate void DatosCambiadosEventHandler(
        object sender, DatosCambiadosEventArgs e);

public event DatosCambiadosEventHandler TextoCambiado;

protected bool OnTextoCambiado(string nuevo,
                               string anterior)
{
  DatosCambiadosEventArgs e =
      new DatosCambiadosEventArgs(nuevo, anterior);
  e.Cancelar = false;
  TextoCambiado(this, e);
  return e.Cancelar;
}
```

Fuente 8. El evento `DatosCambiados` aplicado a una clase derivada de `TextBox`

En nuestro caso, el evento `TextoCambiado` será equivalente al evento `TextChanged` definido en la clase base (`TextBox`) de nuestro control, evento que nosotros aprovechamos para saber cuándo se produce un cambio en el texto. Y para saber cuándo se produce el evento, podemos usar el método asociado al evento que la propia clase base `TextBox` define: `OnTextChanged`. Aunque dentro de ese método no hagamos nada en especial (salvo que queramos volver a asignar el texto que ya hubiera de antes), debemos hacer la llamada al evento `TextoCambiado` para que el cliente se entere de que se está cambiando el texto, y antes de llamar al método `OnTextoCambiado` hay que seguir haciendo la comprobación de que en realidad alguien está interceptando el evento, así evitamos los problemas que comento en el cuadro "De la importancia de comprobar en C# si se interceptan los eventos".

En el código del fuente 9 podemos ver cuál podría ser el código del método `OnTextChanged`, que es más extenso en el código incluido en el ZIP que acompaña al artículo, para que veamos otras posibilidades relacionadas con lo que podemos hacer si el cliente cancela la acción.

Aunque ese cambio también se puede producir al asignar directamente el texto. Por tanto, en la propiedad `Text` también hacemos la correspondiente comprobación, tal como podemos ver en el código del fuente 10.

```
protected override void OnTextChanged(EventArgs e)
{
  // Si no hacemos esta comprobación,
  // no podremos agregar el control al formulario.
  if( TextoCambiado != null )
  {
    // Lanzar el evento, aunque no hagamos nada, por
    // tanto tampoco hace falta saber si se cancela o no
    OnTextoCambiado(this.Text, m_Text);
  }
  // Dejamos que la clase base termine de procesar
  // el evento
  base.OnTextChanged(e);
}
```

Fuente 9. El código del método protegido `OnTextChanged`

```
public override string Text
{
    get { return base.Text; }
    set
    {
        if( TextoCambiado != null )
        {
            if( OnTextoCambiado(value, base.Text) )
            {
                return;
            }
        }
        m_Text = value;
        base.Text = value;
    }
}
```

Fuente 10. En la propiedad `Text` del control también comprobamos si se debe aceptar el cambio

Y ya que estamos con un control personalizado, comentar que podemos usar el atributo `DefaultEventAtribute` para indicar cual será el evento predeterminado de la clase. En nuestro caso, será el even-

De la importancia de comprobar en C# si se interceptan los eventos

Como hemos estado viendo en los ejemplos de esta serie de artículos, cada vez que en C# queremos lanzar un evento, siempre debemos comprobar si ese evento se está interceptando (comprobamos si el evento no es nulo). Tal comprobación no es necesario hacerla en Visual Basic, y esta nota de advertencia va especialmente dirigida a los que antes programaban con Visual Basic (para .NET o anterior) y también para aquellos que quieren empezar a trabajar con C#, ya que si no hacemos esa comprobación y no se está interceptando el evento, será en tiempo de ejecución cuando recibamos el error de que el objeto no está creado; pero la información que recibiremos no será demasiado aclaratoria. Por ejemplo, si ese evento se produce en una clase derivada como es el caso de `MiTextBox`, y se intenta lanzar (sin antes comprobar si se está interceptando) en el método `OnTextChanged`, esto impedirá incluso que ese control lo podamos agregar a un formulario en tiempo de diseño. El diseñador de Windows Forms simplemente producirá un error de que hacemos referencia a un objeto no inicializado y, con suerte (haciendo el trabajo manualmente), detectaremos que el causante de ese error es la propiedad `Text`. En el código de ejemplo que acompaña a este artículo encontrarás una versión "errónea" del control `MiTextBox`.

to que acabamos de definir, tal como vemos en el código del fuente 11. De esa forma, al hacer doble clic en el control, se generará automáticamente ese evento.

```
[DefaultEvent("TextoCambiado")]
public class MiTextBox : TextBox
{
```

Fuente 11. Indicar que evento es el que usará el diseñador de formularios como evento predeterminado

Esto mismo es aplicable al resto de eventos, y no solo a los que nosotros definamos. Por ejemplo, podemos interceptar el evento `MouseMove` tal como vemos en el fuente 13.

```
//private void btnAsignar_MouseMove(object sender, MouseEventArgs e)
private void btnAsignar_MouseMove(object sender, EventArgs e)
{
    MouseEventArgs e2 = (MouseEventArgs)e;
    label1.Text = e2.X + ", " + e2.Y;
}
```

Fuente 13. Con el resto de eventos también podemos usar las clases base como parámetro del método

Usar parámetros de la clase base en los métodos que reciben eventos

Tal como vimos en el artículo dedicado a los delegados (**dotNetManía** nº 30), una de las novedades de C# 2.0 es permitir (por medio de la contravarianza) usar en los delegados parámetros de la clase base del tipo realmente definido en el delegado. Por ejemplo, el delegado que usamos en el evento `TextoCambiado` tiene definido un parámetro del tipo `DatosCambiadosEventArgs` que se deriva de `EventArgs`; pues bien, el método que intercepta ese evento lo podemos definir usando como parámetro o un tipo `DatosCambiadosEventArgs` o de cualquiera que esté en la jerarquía de esa clase, en nuestro caso, el tipo `EventArgs` (y por extensión el tipo `Object`), con lo cual podemos hacer algo como lo mostrado en el código del fuente 12.

Como ya comentamos en su momento, esto no está permitido en Visual Basic ni en las versiones anteriores de C#. Y lo hemos querido aclarar (tanto que en Visual Basic no se puede hacer como que en C# 2.0 sí se puede), porque ahora que desde el pasado mes de noviembre está disponible la nueva versión 3.0 de .NET Framework (antes conocido como WinFX), al interceptar los eventos producidos en los controles usados en *Windows Presentation Foundation* (lo que antes se conocía como Avalon), en realidad el parámetro de esos eventos suele estar derivado de `RoutedEventArgs`, y es muy común que al estar acostumbrados a crear métodos que interceptan eventos en los controles definidos en `Windows.Forms` queramos seguir usando el parámetro `EventArgs`. Si usamos ese tipo de parámetro en vez del "correcto", en C# no pasará nada y todo funcionará bien, pero en Visual Basic recibiremos un error indicándonos que el evento asociado con ese método no tiene la misma firma que el definido en el evento. En realidad, lo que hay detrás de esa nueva característica de los eventos "enrutados" de los controles que usaremos en nuestros proyectos XAML no es tan simple como la forma de usarlos, pero ese es otro tema.

```
void miTextBox1_TextoCambiado(object sender, EventArgs e)
{
    DatosCambiadosEventArgs e2 = (DatosCambiadosEventArgs)e;
    if( string.IsNullOrEmpty(e2.Nuevo) || e2.Nuevo.Length < 4 )
    {
        label1.Text = "¡¡¡ Debes escribir más de tres caracteres !!!";
        label1.Text += "\nAnterior: " + e2.Anterior;
        e2.Cancelar = true;
    }
    else
    {
        label1.Text = miTextBox1.Text;
    }
}
```

Fuente 12. Podemos definir métodos que usen parámetros de clases base del parámetro que el delegado define

Conclusiones

Con este artículo damos por finalizado, al menos por ahora, el tema de los delegados y eventos que hemos estado tratando desde hace tres números. Por supuesto, confiamos que todo lo comentado haya servido para dar al lector una visión clara de cómo usar los eventos y los delegados, al menos desde el punto de vista de la relación tan estrecha que tienen con los eventos.

A pesar de que lo habitual en esta sección de la revista es que el código mostrado y el lenguaje usado sea C#, el número anterior lo dedicamos completamente a Visual Basic, ya que en ese artículo nos centramos en una instrucción exclusiva de la versión 2005 de ese lenguaje. Con esto quiero aclarar que no es que no tengamos en cuenta a los programadores de Visual Basic en esta sección de inicio, ya que como suele ser costumbre, el código que acompaña a estos artículos siempre incluye los ejemplos en ambos lenguajes (al menos cuando hay equivalencia, ya que en ocasiones, algunas de las cosas mostradas solo se pueden hacer en uno de los dos lenguajes). Pero la intención siempre es mostrar los fundamentos de la programación en .NET, y será cosa del lector decidir con qué lenguaje pone en práctica esa información, y siempre intentamos que todo quede explicado de una manera suficientemente clara, cualquiera que sea el lenguaje que el lector decida usar.