Capítulo 3
Los Web Forms

1. Presentación de los Web Forms

Los formularios web (Web Forms) representan la parte más visible de los sitios web ASP.NET y, en consecuencia, la más popular. Se basan en un reparto de responsabilidades de tipo **MVC**: modelo, vista, controlador. Cuando se escribe un formulario utilizando el estilo **código independiente**, la página HTML .aspx se encarga de la representación (vista), la clase C# gestiona los datos y los cálculos realizados con ellos (modelo), mientras que el servidor de aplicaciones ASP.NET coordina el conjunto (controlador). Este análisis resultará familiar, sin duda, a los desarrolladores Java en lo relativo a la organización de sitios web ASP.NET.

Por otro lado, los formularios web son el resultado de la transposición que realiza Microsoft del modelo Visual Basic 6, y una forma original y productiva de desarrollar interfaces gráficas para Internet. El éxito de este modelo ha sido tal, que Sun lo ha replicado por su cuenta en la tecnología de desarrollo web JSF (*Java Server Faces*).

1.1 Estructura de una página ASPX

En el capítulo Los sitios web ASP.NET nos hemos puesto al día con la estructura de una página ASPX desde el punto de vista de la compilación. Ahora se trata de comprender su estructura lógica.

Estudiemos el código que aparece en una página Default.aspx:

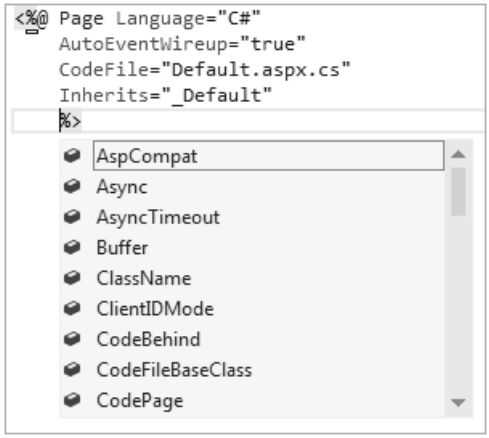
```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    </div>
    </form>
</body>
</html>
```

Este código está formado por tres partes: una directiva page, una declaración de DTD y código XHTML.

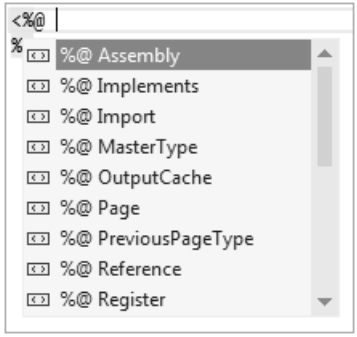
La directiva Page

Las directivas organizan la lectura de una página ASPX en el servidor de aplicaciones. En la página Default.aspx, el atributo **Language** define el lenguaje –C#, VB.NET, C++– utilizado para escribir los scriptlets. Hay otros atributos presentes, que sirven para la comunicación con la página de code behind (**AutoEventWireup**, **CodeFile**, **Inherits**), para aplicar temas, para la gestión de trazas... Descubriremos el uso de estos atributos conforme avance nuestro estudio.

Por suerte, Visual Studio proporciona distintos atributos aplicables utilizando la combinación de teclas [Ctrl][Espacio].



Existen otras directivas disponibles para incluir recursos en el entorno de la página: estrategia de caché, componentes, ensamblados, tipos de página maestra...



Las DTD

Las definiciones de tipo de documento (*Document Type Definition*) las establece el consorcio W3C. Se trata de una norma aplicable a los documentos SGML, XML y HTML que fija las reglas sintácticas y semánticas de la construcción de un documento basado en tags (marcadores).

Los navegadores son bastante tolerantes en lo que respecta a las DTS. Con la versión ASP.NET 1.X, el flujo HTML de salida es compatible con la DTD **HTML transicional de nivel 4**. Salvo el atributo MS_POSITIONNING que no estaba filtrado, el código HTML era completamente estándar. Es cierto que una página ASPX contiene etiquetas especiales (`<asp:label>`, por ejemplo) que se traducen por una secuencia HTML accesible desde el navegador.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

La versión 2.0 aporta una conformidad con **XHTML**, una declaración mucho más estricta del lenguaje HTML. Los puristas pueden dirigirse al sitio web de W3C e introducir una página ASP.NET en el motor de verificación ubicado en la dirección http://validator.w3.org. Las páginas deben estar en conformidad con la DTD correspondiente.

Observación

Preste atención, para realizar esta prueba, es preciso guardar el flujo HTML en un bloc de notas abierto mediante el comando ver código fuente. La función Guardar como - Página HTML del navegador Internet Explorer modifica el archivo y desvirtualiza la prueba.

Para el desarrollador de páginas web, la conformidad con una versión específica del lenguaje HTML no es suficiente para garantizar que una página tenga la misma presentación sea cual sea el navegador. De entrada, los navegadores tienen la responsabilidad de interpretar las reglas de representación tal y como ellos las entiendan. El lenguaje HTML describe el contenido, pero no la representación. Además, las páginas incluyen código JavaScript y estilos CSS, que difieren en su interpretación en función del navegador.

El servidor ASP.NET 2.0 ha introducido otro cambio: desaparece la noción de esquema de navegador de destino. Es cierto que esta directiva no ha podido estar a la par con la evolución de los navegadores, sin contar con la aparición de otros dispositivos de navegación. En su lugar, los sitios web ASP.NET poseen una carpeta **App_Browsers** que considera las características de cada navegador. Este aspecto se estudiará cuando aparezcan los componentes personalizados.

Para ciertos navegadores y programas JavaScript que intervienen en el DOM y que no sean compatibles con la norma XHTML, el servidor de aplicaciones puede configurarse para utilizar el modo HTML transicional. La directiva se ubica en el archivo Web.config:

```
<xhtmlConformance mode="Legacy"/>
```

El atributo mode acepta tres valores:

Legacy	Antiguo formato HTML transicional
Strict	XHTML strict
Transitional	XHTML transicional

El código XHTML

Si bien es cierto que el servidor de aplicaciones ASP.NET 1.X emitía un flujo conforme a la DTD HTML 4 transicional, la propia sintaxis de las páginas ASPX mezclaba secuencias HTML con secuencias XML. Visual Studio 2003 se encargaba de controlar la coherencia del conjunto y generar advertencias cuando era necesario, y el servidor de aplicaciones debía realizar una lectura más atenta (y costosa) para separar las secuencias HTML de las secuencias XML.

Ahora, el elemento <html> contiene una referencia al espacio de nombres XHTML:

```
<html xmlns="http://www.w3.org/1999/xhtml" >
```

Dicho de otro modo, las etiquetas de una página ASPX deben respetar la sintaxis XHTML. De este modo, las etiquetas que comienzan por `asp` (controles web), `uc` (controles de usuario) o `cc` (controles personalizados) no forman parte del vocabulario XHTML. Pero, al menos, la sintaxis es mucho más próxima y más precisa. Y el flujo de salida permanece, en cualquier caso, conforme a la DTD declarada.

Por último, Visual Studio hace todo lo posible para validar de antemano las secuencias HTML que figuran en una página ASPX. Se generan mensajes de advertencia para llamar la atención del desarrollador acerca de las no conformidades.

1.1.1 Estilo anidado, en línea y separado

La organización de una página dinámica es una simple cuestión de estilo. Según la naturaleza de la secuencia HTML que se quiera describir, es preferible optar por la versión anidada o por la versión en línea (inline). Solo el estilo separado (code behind) supone un cambio radical y aporta una distinción neta entre la presentación y el cálculo. Éste es el motivo por el que se le da privilegio en Visual Studio.

El estilo anidado

Son las primeras generaciones de las páginas dinámicas (ASP, PHP) las que imponen el estilo anidado. Con los modelos de componentes web ASP.NET, deja de tener tanto sentido, aunque sigue siendo aplicable. También puede servir en el caso de controles a base de modelos tales como los Repeater o los Data List.

He aquí un ejemplo de código basado en este estilo:

```
<body>
    <form id="form1" runat="server">
    <ul>
    <%
      int i;
       string[] dias = { "lunes", "martes", "miércoles", "jueves",
"viernes", "sábado", "domingo" };
      for(i=0; i< dias.Length; i++)
        {
    %>
            <li><%= dias[i] %></li>
    <%  } %>
    </ul>
    </form>
</body>
```

El desarrollador debe trabajar de la mejor forma para alinear su código como si se tratase de un programa escrito completamente en C#.

El estilo en línea (inline)

El estilo anidado se utiliza, principalmente, en la presentación. No conviene utilizarlo cuando se planifica el procesamiento. La versión en línea separa el código C# y el código HTML en dos partes del mismo archivo .aspx. Las etiquetas `<script runat="server">` indican al compilador que se trata de código C#, aunque puedan reemplazarse por scriptlets `<% %>`.

```
<%@ Page Language="C#" %>
<script runat="server">
    // contiene el código de procesamiento de eventos
    void procesar_click(object sender, EventArgs e)
    {
        mensaje.Text = "¡Ha hecho clic!";
    }

</script>

<!-- límite entre el código C# y el código HTML -->

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Estilo en línea</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:Label ID="mensaje" runat="server"></asp:Label>
        <asp:Button ID="cmd" runat="server" Text="Haga clic aquí"
```

Capítulo 6
Tipos genéricos

1. Introducción

Los tipos genéricos permiten combinar la reutilización del código y la seguridad del tipo. Los tipos genéricos se usan más habitualmente para las colecciones. El Framework .NET expone estas colecciones en el espacio de nombres `System.Collections.Generic`, como el tipo `List<T>` o `Dictionary<TKey, TValue>`. Como es lógico, es posible crear sus propios tipos genéricos para crear una solución adaptada.

La ventaja de los tipos genéricos respecto a las colecciones clásicas como el tipo `ArrayList` es que el tipo de los objetos se conserva, mientras que una colección no genérica almacena los datos, haciendo una conversión al tipo `Object`. La recuperación de un elemento de una colección clásica obliga a hacer la conversión inversa, para hacer corresponder con el tipo esperado, mientras que los tipos genéricos devuelven un objeto ya tipado. A pesar de que la complejidad de codificación es ligeramente superior, los tipos genéricos además aportan mucha más rapidez, sobre todo cuando los elementos de la lista son tipos por valor.

2. La creación de tipos genéricos

Un tipo genérico se define en la declaración de la clase, indicando el parámetro `T`. Este parámetro especifica que el tipo lo elegirá el consumidor de la clase. Puede ser un tipo por valor o por referencia.

Cree una nueva clase `ReportChangeList<T>` en la carpeta **Library** del proyecto:

```
public class ReportChangeList<T>
{
}
```

El parámetro `T` acepta un tipo, que se especificará durante la instanciación:

```
ReportChangeList<int> Ex1 = new ReportChangeList<int>();
ReportChangeList<Object> Ex2 = new ReportChangeList<Object>();
```

Un tipo genérico también puede hacer referencia a varias clases:

```
public class ClaseGenerica<T, U>
```

Los tipos genéricos se pueden sobrecargar, siempre que el número de sus parámetros de tipo no sea idéntico:

```
public class ClaseGenerica<T>
public class ClaseGenerica<T, U>
```

Si dos tipos genéricos tienen el mismo nombre y el mismo número de parámetros de tipo, el compilador devolverá un error:

```
public class ClaseGenerica<T>
public class ClaseGenerica<U>          // No autorizado
```

La clase puede contener miembros que van a utilizar el tipo especificado en la instanciación, gracias al parámetro `T`. Añada el miembro `children` de tipo `List<T>` a la clase:

```
protected List<T> children;
```

Como `List<T>` es un tipo por referencia, se debe instanciar en el constructor para que no sea `null`.

Añada el constructor que instancia la variable `children`:

```
public ReportChangeList()
{
    this.children = new List<T>();
}
```

Durante la instanciación, la clase `ReportChangeList` instanciará una lista genérica con el tipo que se haya especificado.

El objetivo principal de crear un tipo genérico que contenga una lista genérica es poder limitar las funcionalidades expuestas o añadirlas. La clase `ReportChangeList` contendrá las clases hijas en su lista `children` y deberá notificar si el objeto contiene un cambio. Añada la interfaz `IReportChildrenChange` en la declaración de la clase:

```
public class ReportChangeList<T>: IReportChildrenChange
```

Como la clase hereda de la interfaz `IReportChildrenChange`, hay que añadir los miembros definidos en esta interfaz:

```
protected bool hasChanged;

public bool HasChanged
{
    get
    {
        bool result = false;
        foreach (IReportChange child in this.children)
            if (child.HasChanged)
            {
                result = true;
                break;
            }
        return this.hasChanged || result;
    }
    set
    {
        if (this.HasChanged != value)
        {
            this.hasChanged = value;
            if (this.Changed != null)
                this.Changed(this, new EventArgs());
        }
```

```
        if (!value)
        {
            foreach (IReportChange child in this.children)
                child.HasChanged = value;
        }
    }
}

public event EventHandler Changed;

public void ChildChanged(object sender, EventArgs e)
{
    if (this.Changed != null)
        this.Changed(sender, e);
}
```

En el código anterior, los descriptores de acceso de la propiedad `HasChanged` realizan un bucle sobre los elementos de la lista `children` para hacer la actualización de los objetos para el descriptor de acceso `set` o determinar si el objeto, en sí mismo o alguno de los elementos de la lista, se ha modificado por el descriptor de acceso `get`.

Para determinar si el objeto se ha modificado el principio es que si al menos se ha modificado un elemento de la lista o el objeto en sí mismo el objeto entero se considera como modificado. La actualización de la propiedad `HasChanged` de los elementos de la lista solo se lleva a cabo si el objeto recibe el valor `false` en la propiedad `HasChanged`, ya que el objeto se puede modificar pero no sus hijos mientras que, si los hijos se modifican, como consecuencia el objeto padre también se modifica, ya que los hijos forman parte del objeto.

3. Las restricciones de tipo

Como hemos visto en los descriptores de acceso, la interfaz `IReportChange` se usa para acceder a la propiedad `HasChanged` de los elementos de la lista. Esto significa que los elementos añadidos deben implementar obligatoriamente la interfaz. Para limitar los tipos que pueden reemplazar al parámetro genérico `T`, se pueden aplicar restricciones.

A continuación se muestran las restricciones existentes:

Restricción	Descripción
`where T:` *clase*	Restricción sobre la clase base del tipo.
`where T:` *interfaz*	Restricción sobre la interfaz que implementa el tipo.
`where T: class`	El tipo debe ser un tipo por referencia.
`where T: struct`	El tipo debe ser una estructura.
`where T: new()`	El tipo debe tener un constructor sin parámetro.
`where U: T`	El tipo representado por `U` debe ser idéntico al tipo `T`.

Añada una restricción a la interfaz `IReportChange` sobre la clase genérica `ReportChangeList`:

```
public class ReportChangeList<T>: IReportChildrenChange where T:
IReportChange
```

Las restricciones se aplican a todos los parámetros de tipo definidos, ya sea en los métodos o en la definición de tipo.

Las restricciones de tipo también se pueden definir a nivel de los métodos:

```
public void MiMetodo<T>() where T: class
{ }
```

4. Las interfaces genéricas

Como para las listas, sería interesante poder hacer un bucle `foreach` sobre los elementos de nuestro tipo genérico. Para esto es suficiente con implementar la interfaz genérica `IEnumerable<T>`:

```
public class ReportChangeList<T>: IReportChildrenChange,
IEnumerable<T> where T: IReportChange
```

Esta interfaz genérica se comporta como una interfaz clásica, contiene las declaraciones de los miembros necesarios para su implementación y se puede utilizar la clase genérica como referencia.

Añada los miembros `GetEnumerator()` e `IEnumerable.GetEnumerator()` siguientes, para implementar la interfaz en la clase genérica `ReportChangeList`:

```
public IEnumerator<T> GetEnumerator()
{
    for (int i = 0; i < this.children.Count; i++)
    {
        yield return this.children[i];
    }
}
IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
```

La implementación de un enumerador consiste en hacer un bucle sobre los elementos de la lista y devolver cada uno de ellos a la llamada. La palabra reservada `yield return` permite hacer un retorno a la llamada con el valor a devolver en función del valor anterior devuelto. El estado del método se mantiene de tal manera que puede continuar su ejecución hasta la siguiente llamada. El tiempo de vida de este estado está relacionado con el enumerador, el estado del método se elimina cuando la enumeración termina.

4.1 La varianza en las interfaces genéricas

La covarianza y la contravarianza son conceptos utilizados desde la versión 2.0 del Framework .NET. La versión 4 del Framework .NET añade la posibilidad de aplicar esos principios a las interfaces genéricas.

4.1.1 La covarianza

Una interfaz covariante permite a sus miembros devolver tipos más variados que aquellos que se han especificado. La interfaz `IEnumerable<T>`, implementada en la clase `ReportChangeList`, forma parte de las interfaces covariantes. Esto permite reutilizar los métodos que funcionan con las colecciones base genéricas para las colecciones derivadas.

El siguiente ejemplo ilustra la covarianza:

```
public class Clase1
{
    public string Prop1 { get; set; }
    public string Prop2 { get; set; }
    public virtual object Valores()
    {
        return $"{Prop1} {Prop2}";
    }
}
public class Clase2: Clase1

{
    public override string Valores ()
    {
        return $"{Prop1} {Prop2}";
    }
}

static class Program
{
    public static void Write(IEnumerable<Clase1> Elementos1)
    {
        foreach (Clase1 elem in Elementos1)
       {
            Console.WriteLine(elem.Valores());
        }
    }

    [STAThread]
    static void Main()
    {
        IEnumerable<Clase2> Elementos2 = new List<Clase2>();
        Write(Elementos2);
    }
}
```

El método `Write` acepta una colección de tipo `IEnumerable<Clase1>` como parámetro. La covarianza permite llamar al método con una colección de tipo `IEnumerable<Clase2>`, ya que el tipo `Clase2` hereda del tipo `Clase1`.