Introducción a .NET

Jordi Ceballos Villach

PID_00145172

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

La plataforma .NET de Microsoft es actualmente una de las principales plataformas de desarrollo de aplicaciones, tanto de escritorio como para entornos web o dispositivos móviles.

Este módulo pretende dar una visión general de .NET, así como del conjunto de tecnologías que hay a su alrededor como ADO.NET, WinForms, ASP.NET, WPF, WCF, etc., con el objetivo de ofrecer una visión general de la plataforma, y dar a conocer las posibilidades que ofrece.

Todos los ejemplos y capturas de pantalla de esta obra están obtenidos de Visual Studio 2008, que es la versión disponible del IDE de Microsoft en el momento de redacción del presente módulo.

Objetivos

Con el estudio de estos materiales didácticos, alcanzaréis los objetivos siguientes:

1. Obtener una visión general sobre .NET, tanto desde el punto de vista de las tecnologías y productos que lo componen, como sobre las ventajas e inconvenientes respecto a las tecnologías precedentes.

2. Aprender la sintaxis básica del lenguaje C#.

3. Conocer las principales clases disponibles en la biblioteca de clases .NET Framework, que utilizaremos comúnmente en nuestros desarrollos, tanto si se trata de aplicaciones de escritorio, web u otros.

4. Conocer ADO.NET y LINQ, que nos permiten acceder a las fuentes de datos (bases de datos, archivos XML, etc.).

5. Conocer las posibilidades de las tecnologías Windows Forms, ASP.NET y WPF, que nos permiten crear las interfaces de usuario de nuestras aplicaciones.

6. Obtener un conocimiento básico sobre el desarrollo de aplicaciones para dispositivos móviles.

1. La plataforma .NET

En este apartado se introduce la plataforma .NET, su arquitectura general, así como los productos y tecnologías que lo componen.

1.1. Orígenes y características de .NET

En el año 2000 Microsoft presentó la plataforma .NET, con el objetivo de hacer frente a las nuevas tendencias de la industria del software, y a la dura competencia de la plataforma Java de Sun.

Plataforma

Una plataforma es un conjunto de tecnologías, junto con un entorno integrado de desarrollo (IDE) que permiten desarrollar aplicaciones.

.NET es una plataforma para el desarrollo de aplicaciones, que integra múltiples tecnologías que han ido apareciendo en los últimos años como ASP.NET, ADO.NET, LINQ, WPF, Silverlight, etc., junto con el potente entorno integrado de desarrollo Visual Studio, que permite desarrollar múltiples tipos de aplicaciones.

Por ejemplo, se pueden desarrollar las siguientes aplicaciones:

- Aplicaciones de línea de comandos.
- Servicios de Windows.
- Aplicaciones de escritorio con Windows Forms o WPF.
- Aplicaciones web con el *framework* ASP.NET, o Silverlight.
- Aplicaciones distribuidas SOA mediante servicios web.
- Aplicaciones para dispositivos móviles con Windows Mobile.

Framework

Un *framework* es un conjunto de bibliotecas, compiladores, lenguajes, etc., que facilitan el desarrollo de aplicaciones para una cierta plataforma.

Microsoft sólo ofrece soporte .NET para sistemas operativos Windows y las nuevas generaciones de dispositivos móviles. Respecto al resto de plataformas, el proyecto Mono[1] (llevado a cabo por la empresa Novell) ha creado una implementación de código abierto de .NET, que actualmente ofrece soporte completo para Linux y Windows, y soporte parcial para otros sistemas operativos como por ejemplo MacOS.

(1)Más información sobre el proyecto Mono en www.mono-project.com.

Los elementos principales de la plataforma .NET son:

- **.NET Framework**: es el núcleo de la plataforma, y ofrece la infraestructura necesaria para desarrollar y ejecutar aplicaciones .NET.

- **Visual Studio y Microsoft Expression**: conforman el entorno de desarrollo de Microsoft, que permite desarrollar cualquier tipo de aplicación .NET (ya sea de escritorio, web, para dispositivos móviles, etc.). En Visual Studio, el programador puede elegir indistintamente entre diversos lenguajes como C# o Visual Basic .NET, y en todos ellos se puede hacer exactamente lo mismo, con lo que a menudo la elección es simplemente debida a las preferencias personales de cada programador.

Preferencias de software

Debido a la similitud entre lenguajes, a menudo los programadores con experiencia Java eligen programar en C#, mientras que los programadores Visual Basic se decantan mayoritariamente por Visual Basic .NET.

1.1.1. Ventajas e inconvenientes de .NET

Las principales ventajas de .NET son las siguientes:

- **Fácil desarrollo de aplicaciones**: en comparación con la API Win32 o las MFC, las clases del .NET Framework son más sencillas y completas.

- **Mejora de la infraestructura de componentes**: la anterior infraestructura de componentes lanzada en 1993 (componentes COM) tenía algunos inconvenientes (se tenían que identificar de forma única, era necesario registrarlos, etc.).

- **Soporte de múltiples lenguajes**: .NET no sólo ofrece independencia del lenguaje (ya lo ofrecía COM), sino también integración entre lenguajes. Por ejemplo, podemos crear una clase derivada de otra, independientemente del lenguaje en que ésta haya sido desarrollada. Los lenguajes más utilizados de la plataforma .NET son C# y Visual Basic .NET, aunque existen muchos otros.

Otros lenguajes

Existen multitud de lenguajes adicionales como, por ejemplo, C++, F#, Cobol, Eiffel, Perl, PHP, Python o Ruby.

- **Despliegue sencillo de aplicaciones**: .NET regresa a las instalaciones de impacto cero sobre el sistema, donde sólo hay que copiar una carpeta con los archivos de la aplicación para "instalarla". Aunque sigue siendo posible, la mayoría de aplicaciones .NET no hacen uso del registro de Windows, y guardan su configuración en archivos XML.

- **Solución al infierno de las DLL**: permite tener diferentes versiones de una DLL al mismo tiempo, y cada aplicación carga exactamente la versión que necesita.

El infierno de las DLL

"DLL Hell" fue durante años un grave problema de Windows, ya que no permitía tener al mismo tiempo diferentes versiones de una misma DLL, y esto provocaba que ciertos programas no funcionaran, al requerir una versión concreta de DLL.

Como inconvenientes de .NET, podemos destacar los siguientes:

- **Reducido soporte multiplataforma**: Microsoft sólo ofrece soporte para entornos Windows. El proyecto Mono[2], liderado por Miguel de Icaza, ha portado .NET a otras plataformas como Linux o Mac OS X, pero su soporte es limitado.

[(2)]Más información sobre el proyecto Mono en www.mono-project.com.

- **Bajo rendimiento**: debido a que el código .NET es en parte interpretado, el rendimiento es menor en comparación con otros entornos como C/C++,

que son puramente compilados. De hecho, para ser precisos, el código .NET es en primer lugar compilado por Visual Studio durante el desarrollo, y posteriormente interpretado por el Common Language Runtime en el momento de su ejecución.

- **Decompilación**[3]: igual como ocurre con Java, las aplicaciones .NET contienen la información necesaria que permitiría a un *hacker* recuperar el código fuente del programa a partir de los ficheros compilados. Para evitarlo, podemos aplicar técnicas de ofuscación sobre el código fuente, de forma que su comportamiento sigue siendo el mismo, pero al estar el código ofuscado, complicamos la reingeniería inversa de la aplicación.

(3)Consultad el decompilador Salamander de RemoteSoft.

1.1.2. Evolución de .NET

Desde la aparición de la primera versión estable de .NET en el 2002, Microsoft ha continuado añadiendo funcionalidades a la plataforma y mejorando sus herramientas de desarrollo.

A continuación, veremos las diferentes versiones de .NET existentes:

a) .NET Framework 1.0: la primera versión del .NET Framework apareció en el 2002, junto con Visual Studio .NET 2002, el nuevo entorno de desarrollo de Microsoft.

b) .NET Framework 1.1: la versión 1.1 aparece en el 2003, junto con Visual Studio .NET 2003 y el sistema operativo Windows Server 2003. Por primera vez aparece .NET Compact Framework, que es una versión reducida del .NET Framework, diseñada para su ejecución en dispositivos móviles.

c) .NET Framework 2.0: aparece en el 2005, junto con Visual Studio 2005 (la palabra .NET desaparece del nombre del producto) y SQL Server 2005 (la nueva versión del motor de bases de datos de Microsoft, después de 5 años). Esta versión incluye cambios sustanciales en los lenguajes .NET, como son los tipos genéricos o los tipos abstractos. También aparece una segunda versión del .NET Compact Framework.

d) .NET Framework 3.0: aparece en el 2006, junto con Windows Vista. La gran novedad en esta versión son las siguientes tecnologías:

- Windows Presentation Foundation (WPF): para el desarrollo de interfaces gráficas avanzadas, con gráficos 3D, vídeo, audio, etc.
- Windows Communication Foundation (WCF): para el desarrollo de aplicaciones SOA orientadas a servicios.
- Windows Workflow Foundation (WWF): facilita la creación de flujos de trabajo que se pueden ejecutar desde una aplicación.

- Windows CardSpace: permite almacenar la identidad digital de una persona y su posterior identificación.

e) .NET Framework 3.5: aparece a finales del 2007, junto con Visual Studio 2008, SQL Server 2008 y Windows Server 2008. Esta nueva versión añade LINQ para el acceso a bases de datos, así como múltiples novedades en el entorno de desarrollo (Javascript intellisense, posibilidad de desarrollar para diferentes versiones del .NET Framework, etc.).

1.2. Visual Studio

Microsoft Visual Studio es un entorno integrado de desarrollo (IDE) compartido y único para todos los lenguajes .NET. El entorno proporciona acceso a todas las funcionalidades del .NET Framework, así como a muchas otras funcionalidades que hacen que el desarrollo de aplicaciones sea más ágil.

1.2.1. Evolución de Visual Studio

Visual Studio no es un producto nuevo; ya existía antes de la aparición de .NET, para desarrollar aplicaciones mediante las tecnologías anteriores. Existían diferentes versiones del producto para cada uno de los lenguajes, básicamente C++, Visual Basic y J#, aparte de la versión completa que daba soporte a todos ellos en el mismo entorno de trabajo. La última versión antes de la aparición de .NET es la 6.0.

En el 2002, con la aparición de la versión 1.0 de .NET, se cambió el nombre del producto por **Visual Studio .NET 2002**, aunque internamente esta versión correspondía con la versión 7.0.

En el 2005, apareció la versión **Visual Studio 2005** (8.0), ya sin la palabra .NET en el nombre del producto. Esta versión, aparte de proporcionar las nuevas funcionalidades de la versión 2.0 del .NET Framework, se integra con el servidor de bases de datos SQL Server 2005, que apareció al mismo tiempo.

En el 2008, apareció **Visual Studio 2008** (9.0), con las novedades de la versión 3.5 del .NET Framework, e integrada con SQL Server 2008.

1.2.2. Ediciones de Visual Studio

Visual Studio 2008 se presenta en diferentes ediciones:

- **Edición Express**: edición con funcionalidades limitadas, diseñada para desarrolladores principiantes. Se puede descargar y utilizar gratuitamente, siempre y cuando no sea con fines lucrativos. Existen versiones express de C#, Visual Basic .NET, C++.NET, Web Developer y SQL Server.

- **Edición Standard**: permite desarrollar aplicaciones Windows o ASP.NET en cualquiera de los lenguajes de .NET.

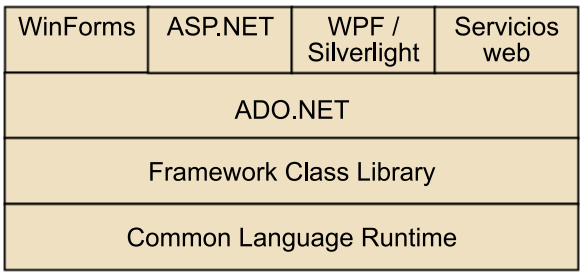
- **Edición Profesional**: edición pensada para desarrolladores profesionales. Permite desarrollar aplicaciones para dispositivos móviles, y aplicaciones basadas en Microsoft Office.

- **Edición Team System**: recomendada para empresas con equipos de trabajo, y consta de varias versiones específicas para cada una de las funciones dentro de un equipo de desarrollo: arquitecto, desarrollador, *tester*, etc. El producto Visual Studio Team Suite es una versión especial que incorpora todas las funcionalidades de los productos Team System.

1.3. Arquitectura de .NET

El *Common Language Runtime* (CLR) es el entorno de ejecución de .NET, que incluye una máquina virtual, análoga en muchos aspectos a la máquina virtual de Java. El CLR se encarga de ofrecer el entorno donde se ejecutan las aplicaciones .NET y, por tanto, se encarga de activar los objetos, ejecutarlos, gestionar la memoria, realizar comprobaciones de seguridad, etc.

Figura 1. Arquitectura de .NET



Por encima del CLR se sitúa la Framework Class Library, que con más de 4000 clases es una de las mayores bibliotecas de clases existentes. En el siguiente nivel, están las clases que permiten el acceso a datos por medio de ADO.NET. Y en la última capa, están las tecnologías para la creación de aplicaciones, que son las siguientes:

- **Win Forms**. Desarrollo de aplicaciones de escritorio.
- **ASP.NET**. Desarrollo de aplicaciones web. Es la evolución de ASP.
- **WPF**. Nueva tecnología para el desarrollo de aplicaciones de escritorio.
- **Silverlight**. Subconjunto de WPF destinado al desarrollo de aplicaciones web. Es una tecnología similar a Flash de Adobe.
- **Servicios web**. Desarrollo de aplicaciones distribuidas.

1.3.1. Compilación y MSIL

Al compilar una aplicación .NET obtenemos archivos con extensión .exe o .dll, pero no debemos confundirnos y pensar que contienen código máquina, sino el código intermedio MSIL[4]. El objetivo de MSIL es el mismo que los *bytecodes* de Java, es decir, disponer de un código intermedio universal (no ligado a ningún procesador), que pueda ser ejecutado sin problemas en cualquier sistema que disponga del intérprete correspondiente.

(4)Del inglés, *Microsoft intermediate language*.

En .NET, la ejecución está basada en un compilador JIT que, a partir del código MSIL, va generando el código nativo bajo demanda, es decir, compila las funciones a medida que se necesitan. Como una misma función puede ser llamada en diversas ocasiones, el compilador JIT, para ser más eficiente, almacena el código nativo de las funciones que ya ha compilado anteriormente.

Compilación *just in time* (JIT)

La compilación *just in time* hace referencia a la compilación del código intermedio MSIL que tiene lugar en tiempo de ejecución. Se denomina *just in time* ("justo a tiempo") porque compila las funciones justo antes de que éstas se ejecuten.

1.3.2. Ensamblados

Un ensamblado[5] es la unidad mínima de empaquetado de las aplicaciones .NET, es decir, una aplicación .NET se compone de uno o más ensamblados, que acaban siendo archivos .dll o .exe.

(5)Del inglés, *assembly*.

Los ensamblados pueden ser privados o públicos. En el caso de ser privados, se utilizan únicamente por una aplicación y se almacenan en el mismo directorio que la aplicación. En cambio, los públicos se instalan en un directorio común de ensamblados llamado *global assembly cache* o GAC, y son accesibles para cualquier aplicación .NET instalada en la máquina.

Aparte del código MSIL, un ensamblado puede contener recursos utilizados por la aplicación (como imágenes o sonidos), así como diversos metadatos que describen las clases definidas en el módulo, sus métodos, etc.

2. El lenguaje C#

Aunque la plataforma .NET permite programar con múltiples lenguajes, hemos seleccionado C# por ser uno de los más representativos y utilizados de .NET. En este apartado, se presenta una introducción a la sintaxis del lenguaje C# que, tal y como se podrá observar, presenta muchas similitudes con lenguajes como C++ y Java.

2.1. Una introducción al lenguaje C#

La mejor forma de empezar a aprender un lenguaje de programación nuevo suele ser analizando algún ejemplo de código sencillo. Para no romper la tradición, empecemos con el típico *HolaMundo*:

```
namespace HolaMundo
{
  class HolaMundo
  {
    static void Main(string[] args)
    {
      Console.WriteLine ("Hola Mundo");
    }
  }
}
```

Analicemos los elementos del programa anterior:

- **Definición del *namespace***: todas las clases están incluidas dentro de un espacio de nombres concreto, que se indica dentro del código fuente mediante la instrucción *namespace*.

- **Definición de la clase**: la forma de definir una clase es con la palabra clave *class*, seguida del nombre que queremos dar a la clase, y todos los elementos e instrucciones que pertenecen a la definición de la clase entre llaves. Cabe destacar que en C# todo son clases, con lo que todas las líneas de código deben estar asociadas a alguna clase.

- **El método *main***: punto de entrada a la aplicación, por donde empieza su ejecución. Puede recibir un *array* llamado *args*, que contiene los parámetros pasados al ejecutable al lanzar su ejecución.

Los tipos de datos de C# se dividen en dos categorías:

1) Tipos valor

Son los tipos primitivos del lenguaje: enteros, reales, caracteres, etc. Los tipos valor más usuales son:

Tipo	Descripción
short	Entero con signo de 16 bits
int	Entero con signo de 32 bits
long	Entero con signo de 64 bits
float	Valor real de precisión simple
double	Valor real de precisión doble
char	Carácter Unicode
bool	Valor booleano (true/false)

2) Tipos referencia

Los tipos referencia nos permiten acceder a los objetos (clases, interfaces, *arrays*, *strings*, etc.). Los objetos se almacenan en la memoria *heap* del sistema, y accedemos a ellos a través de una referencia (un puntero). Estos tipos tienen un rendimiento menor que los tipos valor, ya que el acceso a los objetos requiere de un acceso adicional a la memoria *heap*.

2.2. Sintaxis de C#

La sintaxis de un lenguaje es la definición de las palabras clave, los elementos y las combinaciones válidas en ese lenguaje. A continuación, describimos de forma resumida la sintaxis de C#.

2.2.1. *Case sensitive*

C# es un lenguaje *case sensitive*, es decir, que diferencia entre mayúsculas y minúsculas. Por ejemplo, si escribimos *Class* en lugar de *class*, el compilador dará un error de sintaxis.

2.2.2. Declaración de variables

En C# todas las variables se tienen que declarar antes de ser utilizadas, y se aconseja asignarles siempre un valor inicial:

```
int prueba = 23;
float valor1 = 2.5, valor2 = 25.0;
```

2.2.3. Constantes

Una constante es una variable cuyo valor no puede ser modificado, es decir, que es de sólo lectura. Se declaran con la palabra clave *const*:

```
const int i = 4;
```

2.2.4. *Arrays*

Un *array* permite almacenar un conjunto de datos de un mismo tipo, a los que accedemos según su posición en el *array*. En la siguiente línea, declaramos un *array* que almacena 5 enteros, y guardamos un primer valor:

```
int[] miArray = new int[5];
miArray[0] = 10;
```

2.2.5. Comentarios

Los comentarios son imprescindibles si pretendemos escribir código de calidad. En C# existen tres tipos de comentarios:

Comentarios	
De una sola línea	`// Ejemplo de comentario`
De múltiples líneas	`/* ... */`
	Cualquier carácter entre el símbolo /* y el símbolo */, se considera como parte del comentario, aunque abarque varias líneas.
Comentarios XML	`///`

Lectura recomendada

Recomendamos la lectura del libro *Code Complete*, de Steve McConnell, una excelente guía para escribir código de calidad. Contiene un interesante capítulo dedicado a los comentarios.

Incorporamos *tags* XML documentando el código, y así luego podemos generar un archivo XML con toda la documentación del código:

```
/// <summary> Breve descripción de una clase </summary>
/// <remarks> Ejemplo de uso de la clase</remarks>
public class MiClase() { ... }
```

Una vez tenemos el código comentado con *tags* XML, podemos generar una salida elegante en formato HTML o CHM con la herramienta SandCastle de Microsoft.

2.2.6. Visibilidad

Podemos restringir el acceso a los datos y métodos de nuestras clases, indicando uno de los siguientes modificadores de acceso:

- ***public***: sin restricciones.
- ***private***: accesible sólo desde la propia clase.
- ***protected***: accesible desde la propia clase, y desde clases derivadas.
- ***internal***: accesible desde el propio ensamblado.

Ejemplo:

```
class Prueba {
  public int cantidad;
  private bool visible;
}
```

2.2.7. Operadores

En C# podemos hacer uso de los siguientes operadores:

Operadores	
Aritméticos	+, –, *, /, % (módulo)
Lógicos	& (AND bit a bit), | (OR bit a bit), ~ (NOT bit a bit)
	&& (AND lógico), || (OR lógico), ! (NOT lógico)
Concatenación de cadenas de caracteres	+
Incremento / decremento	++, – –
Comparación	==, != (diferente), <, >, <=, >=
Asignación	=, +=, –=, *=, /=, %=, &=, |=, <<=, >>= Las combinaciones +=, –=, *=, etc., permiten asignar a una variable el resultado de realizar la operación indicada. Ejemplo: x += 3 es equivalente a $x = x + 3$

2.2.8. Enumeraciones

Una enumeración es una estructura de datos que permite definir una lista de constantes y asignarles un nombre. A continuación, mostramos una enumeración para los días de la semana:

```
enum DiaSemana
{
  Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo
}
```

De ese modo el código es mucho más legible ya que, en vez de utilizar un entero del 1 al 7 para representar el día, se utiliza una constante y un tipo específico de datos para el día de la semana, de forma que se evitan problemas adicionales como, por ejemplo, controlar que el valor de una variable que represente el día esté entre 1 y 7.

2.2.9. Estructuras

Una estructura contiene diversos elementos que pueden ser de diferentes tipos de datos:

```
struct Punto
{
  int x;
  int Y;
  bool visible;
}
```

Las estructuras se declaran igual que cualquier tipo por valor:

```
Punto p;
p.x = 0;
p.y = 0;
```

2.2.10. Control de flujo

El lenguaje C# incorpora las siguientes sentencias de control de flujo:

- ***if***: sentencia condicional. Un ejemplo típico sería el siguiente:

```
if (x <= 10)
{
  // Sentencias a ejecutar si la condición es cierta
}
else
{
  // Sentencias a ejecutar si la condición es falsa
}
```

- ***switch***: la instrucción *switch* es una forma específica de instrucción condicional, en la que se evalúa una variable, y en función de su valor, se ejecuta un bloque u otro de instrucciones. Ejemplo:

```
switch (var)
{
  case 1: // sentencias a ejecutar si var es 1
          break;
```

```
  case 2: // sentencias a ejecutar si var es 2
          break;
  ...

  default: // sentencias a ejecutar
           // en caso de que ninguna de las
           // condiciones "case" se cumplan
           break;
}
```

- ***for***: permite ejecutar un bloque de código un cierto número de veces. El siguiente ejemplo ejecuta el bloque de instrucciones 10 veces:

```
for (int i = 0; i<10; i++)
{
  // sentencias a ejecutar
}
```

- ***while***: el siguiente ejemplo es equivalente al anterior del *for*.

```
int i = 0;
while (i<10)
{
     // sentencias a ejecutar
     i++;
}
```

- ***do-while***: igual que el anterior, excepto que la condición se evalúa al final. La diferencia fundamental es que, mientras que en un bucle *for* o *while*, si la condición es falsa de entrada, no se ejecuta ninguna iteración, en un bucle *do-while* siempre se ejecuta como mínimo una iteración.

```
do
{
  // sentencias a ejecutar
}
while (condición);
```

- ***foreach***: permite recorrer todos los elementos de una colección desde el primero al último. La sintaxis es la siguiente:

```
foreach (tipo nombre_variable in colección)
{
  // sentencias a ejecutar
}
```

Un ejemplo de colección son los *arrays*:

```
int [] nums = new int [] { 4,2,5,7,3,7,8 };
foreach (int i in nums)
{
  Console.WriteLine (i);
}
```

2.2.11. Paso de parámetros

En C# el paso de parámetros a una función se puede realizar por valor, por referencia, o ser un parámetro de salida.

Paso de parámetros por valor

En primer lugar, recordemos que los tipos de datos que pasemos como parámetro pueden ser tipos valor (tipos primitivos), o tipos referencia (objetos), ya que analizaremos ambos casos por separado.

Cuando pasamos un tipo valor como parámetro a una función, éste se pasa por valor, es decir, se copia el valor de la variable al método, en la variable local representada por el parámetro. De ese modo, si se modifica el valor de las variables de los parámetros, no se modifica el valor de las variables que se pasaron inicialmente como parámetro. En el siguiente ejemplo, una vez ejecutado el método *PasoDeParametros*, la variable a seguirá valiendo 0.

```
static void PasoDeParametros (int param)
{
  param = 5;
}

static void Main(string[] args)
{
  int a = 0;
  PasoDeParametros(a);
}
```

En el caso de los tipos referencia, cuando se pasan como parámetro a una función, lo que se pasa realmente es su dirección de memoria. Por tanto, al acceder al parámetro dentro del método, se modifica la variable original. Sin embargo, lo que no se puede hacer es cambiar la dirección de memoria a la que apunta la variable original.

Excepción

El *string* es una excepción, ya que al pasar un *string* como parámetro se copia la cadena, es decir, se comporta como un tipo valor.

En el siguiente ejemplo, una vez ejecutado el método *PasoDeParametros*, la posición 0 del vector a pasará a tener valor 5, y aunque en la función se le asigna un nuevo *array* con valores negativos, esto no tiene ninguna relevancia al salir de la función.

```
static void PasoDeParametros (int[] param)
{
  param[0] = 5;
  param = new int[4] { -1, -2, -3, -4};
}

static void Main(string[] args)
{
  int[] a = { 0, 1, 2, 3 };
  PasoDeParametros(a);
}
```

Paso de parámetros por referencia

El paso de parámetros por referencia lo analizaremos primero con tipos valor, y a continuación con tipos referencia.

Para pasar un tipo valor como parámetro por referencia a una función, lo indicamos con la palabra clave *ref*. En este caso, la función recibe la dirección de memoria de la variable, con lo que si la modifica, el cambio tiene lugar también en la variable original. En el siguiente ejemplo, una vez ejecutado el método *PasoDeParametros*, la variable a pasará a valer 5.

```
static void PasoDeParametros(ref int param)
{
  param = 5;
}
static void Main(string[] args)
{
  int a = 0;
  PasoDeParametros(ref a);
}
```

Para pasar un tipo referencia como parámetro por referencia a una función, lo indicamos con la palabra clave *ref*. En este caso, la función recibe la dirección de memoria donde está almacenada la variable original, con lo cual podemos tanto modificar los valores de la variable original, como también hacer que la variable original acabe apuntando a una dirección diferente.

En el siguiente ejemplo, una vez ejecutado el método *PasoDeParametros*, la posición 0 del vector a pasará a tener valor –1, ya que realmente hemos hecho que apunte a la dirección de memoria del nuevo *array*.

```
static void PasoDeParametros(ref int[] param)
{
  param[0] = 5;
  param = new int[4] { -1, -2, -3, -4 };
}
static void Main(string[] args)
{
  int[] a = { 0, 1, 2, 3 };
  PasoDeParametros(ref a);
  Console.WriteLine(a[0]);
}
```

Parámetros de salida

Por último, también existen los parámetros de salida, que se indican con la palabra clave *out*, y sólo sirven para devolver valores en un método.

En el siguiente ejemplo, la función devuelve los valores $a = 5$ y $b = 10$. En caso de que la función hubiera intentado leer los valores de a y b antes de asignarles ningún valor, hubiéramos obtenido un error de compilación.

```
static void PasoDeParametros(out int a, out int b)
{
  a = 5;
  b = 10;
}

static void Main(string[] args)
{
  int a, b;
  PasoDeParametros(out a, out b);
}
```

2.2.12. Sobrecarga de métodos

La sobrecarga de métodos permite tener múltiples métodos con el mismo nombre, aunque con parámetros distintos. Al llamar al método, se invoca a uno u otro en función de los parámetros de la llamada. Veamos un ejemplo:

```
public static int prueba(int a)
{
  return 2 * a;
}
```

```
public static int prueba(int a, int b)
{
  return a + b;
}

static void Main(string[] args)
{
  Console.WriteLine(prueba(10));
  Console.WriteLine(prueba(10, 40));
}
```

2.3. Programación orientada a objetos con C#

C# es un lenguaje orientado a objetos. A continuación, veremos los conceptos básicos de orientación a objetos y su utilización en C#.

2.3.1. Definición de clases

En C# podemos definir una clase mediante la palabra clave *class*:

```
class MiClase
{
  public int valor; // Miembro de datos

  public int calculo() // Miembro de función
  {
    return valor*2;
  }
}
```

En la definición de la clase, especificamos los miembros de datos (variables o propiedades) que describen el estado del objeto, y un conjunto de operaciones que definen su comportamiento (funciones o métodos).

Existe la posibilidad de definir una clase en varios ficheros de código fuente, lo que se denominan clases parciales. Si creamos un formulario en Visual Studio, automáticamente se crea una clase parcial repartida en dos archivos; en uno de ellos escribimos el código asociado a los eventos, y en el otro Visual Studio genera automáticamente el código de la interfaz de usuario.

A modo de ejemplo, si creamos un formulario llamado *Form1*, obtenemos una clase parcial llamada *Form1* repartida en los archivos:

- **Form1.cs**: en esta clase, programaremos todo el código de los eventos del formulario que deseemos implementar (al cargar la página, al pulsar un botón, etc.).

- **Form1.Designer.cs**: esta clase nunca la modificaremos manualmente, sino que será Visual Studio el que generará el código correspondiente al formulario que diseñemos mediante el Visual Studio.

2.3.2. Objetos

Un objeto es una instancia concreta de una clase. La sintaxis para la instanciación de un objeto es la siguiente:

```
MiClase obj;
obj = new MiClase ();
obj.valor = 3;
```

El caso de no querer hacer más uso de un objeto, podemos asignarle el valor *null*. De esta forma, el recolector de basura de C# detectará que el objeto ha dejado de ser referenciado y, por tanto, liberará la memoria que utiliza.

2.3.3. Propiedades

Una propiedad permite encapsular una variable junto con los métodos que permiten consultar o modificar su valor. En otros lenguajes, no existen las propiedades, con lo que para cada variable de una clase se añade manualmente los métodos *get*/*set* correspondientes.

En el siguiente ejemplo, definimos la propiedad *Descuento*, cuyo valor no permitimos que pueda ser negativo:

```
public class MiClase
{
  private int descuento;
  public int Descuento
  {
    get
    {
      return descuento;
    }
    set
    {
      if (value > 0) descuento = value;
      else descuento = 0;
    }
  }
}
```

Una vez definida la propiedad, su uso es muy sencillo:

```
MiClase c = new MiClase();
```

```
c.Descuento = 10;
Console.WriteLine(c.Descuento);
```

2.3.4. Construcción de objetos

Un constructor es el método que nos permite crear objetos de una cierta clase, y es útil porque permite asegurar que los objetos se inicializan correctamente, con lo que aseguramos la integridad de las instancias. Es habitual sobrecargar los constructores para disponer de múltiples constructores en función del número de parámetros recibidos. En el siguiente ejemplo, la clase tiene un constructor por defecto (sin parámetros), y por otro lado un constructor con un parámetro:

```
class MiClase
{
  public int valor;
  public MiClase()
  {
    valor = 0;
  }
  public MiClase(int valor)
  {
    this.valor = valor;
  }
}
```

En el ejemplo anterior hemos utilizado la palabra clave *this*, que permite acceder a los miembros de datos de la propia clase. En caso de no haber utilizado *this*, deberíamos haber puesto un nombre diferente al parámetro del constructor para que no coincidiera con el nombre de la variable.

Al tener dos constructores, podemos crear objetos de dos formas distintas:

```
MiClase obj1, obj2;
obj1 = new MiClase();
obj2 = new MiClase(10);
```

2.3.5. Destrucción de objetos

Los métodos destructores permiten liberar los recursos que ha utilizado el objeto a lo largo de su ejecución, pero que a partir de un cierto momento han dejado de ser utilizados y, por tanto, están esperando ser liberados, como por ejemplo sucedería con una conexión a base de datos.

Un ejemplo de destructor sería el siguiente:

```
~MiClase()
```

```
{
  Console.WriteLine("El objeto ha sido destruido");
}
```

El carácter ~

El carácter ~ normalmente no aparece en los teclados españoles, pero podemos hacerlo aparecer pulsando Alt 126 (en el teclado numérico).

La liberación de recursos tiene lugar mediante el proceso de recolección automática de basura[6], que será el encargado, entre otras cosas, de ejecutar los destructores de los objetos que no vayan a ser utilizados más. Eso sí, la liberación de recursos no se produce inmediatamente. De hecho, no podemos predecir cuándo se ejecutará el *Garbage Collector* y, hasta podría llegar a suceder que éste no se ejecute hasta que haya finalizado la ejecución de nuestro programa.

(6)Del inglés, *garbage collection*.

2.3.6. Métodos estáticos

Un método estático es aquel que podemos invocar sin necesidad de crear previamente un objeto de la clase que lo contiene. Como ejemplo, mostramos el método *Pow* de la clase *Math*:

```
double resultado = Math.Pow(2.0, 8);
```

Otro ejemplo de método estático es el método *Main*, que no se ejecuta sobre ninguna instancia de clase, ya que al ser el primer método que se ejecuta, aún no se ha creado ningún objeto.

2.3.7. Herencia

Las clases pueden organizarse en jerarquías que permiten extender y reutilizar las funcionalidades de unas clases en otras. Estas jerarquías se crean mediante la relación de herencia, que relaciona dos clases: la superclase y la subclase, donde una subclase hereda automáticamente toda la funcionalidad de la superclase.

```
class MiClaseHija : MiClasePadre
{
}
```

Si al definir una clase no especificamos su clase padre, por defecto dicha clase hereda de la clase *Object*, que contiene algunos métodos comunes a todas las clases como, por ejemplo, *Equals* o *ToString*.

En C# una clase sólo puede heredar de una única clase padre. En cambio, hay lenguajes que permiten la herencia múltiple, es decir, que las clases pueden heredar de múltiples clases padre al mismo tiempo.

2.3.8. Interfaces

Una interfaz es una especie de plantilla que especifica cómo han de ser un conjunto de métodos (indicando sus parámetros y valores de retorno), pero sin disponer de su implementación.

Una vez definida una interfaz, habrá clases que la implementarán y que, por tanto, estarán obligadas a implementar todos aquellos métodos indicados en la interfaz. De hecho, en las interfaces no hay problemas con la herencia múltiple, de manera que una clase puede implementar sin problemas múltiples interfaces al mismo tiempo (en caso de haber métodos de las clases padre que coincidan, no hay problema, ya que estos métodos sólo tendrán una única implementación que será la definida en la propia clase).

En el siguiente ejemplo, definimos la interfaz *IPrueba* y la clase *MiClase* implementa dicha interfaz, con lo que se ve obligada a implementar el método *calculo* (en caso de no hacerlo, el código no compilaría):

```
public interface IPrueba
{
  double calculo(double x);
}

public class MiClase : IPrueba
{
  public double calculo(double x)
  {
    return Math.Pow(x, 3);
  }
}
```

2.3.9. Sobrescritura de métodos

Cuando una clase hereda de otra, puede sobrescribir los métodos marcados como virtuales, y al sobrescribirlos lo ha de indicar explícitamente con el modificador *override*. A modo de ejemplo, la clase *Object* ofrece el método virtual *ToString*, que normalmente sobrescribimos en nuestras clases para que, al llamar a dicho método, se llame a nuestra propia implementación del método, y no al de la clase base. En caso de que quisiéramos llamar al método de la clase base, lo haríamos con la sentencia *base.ToString*().

En el siguiente ejemplo, creamos una clase Persona que implementa su propio método *ToString* y, en el método *Main*, se mostrará por pantalla el nombre de la persona. Cuando el método *Console.WriteLine* no recibe como parámetro un *String*, automáticamente llama al método *ToString*() del objeto que le hayamos pasado como parámetro.

```
class Persona
{
  String nombre;
  public Persona(String nombre)
  {
    this.nombre = nombre;
  }
  public override string ToString()
  {
    return nombre;
  }
}

class Program
{
  static void Main(string[] args)
  {
    Persona p = new Persona("Luis");
    Console.WriteLine(p);
  }
}
```

2.3.10. Genéricos

Los tipos genéricos son clases que pueden aplicarse sobre distintos tipos de datos. Para hacerlo utilizan un tipo de dato comodín, que puede tomar la forma de cualquier tipo de datos. En el siguiente ejemplo, la propiedad *e1* permite almacenar un objeto de cualquier clase:

```
public class Prueba<T>
{
  private T e1;
  public T E1
  {
    get { return e1; }
    set { e1 = value; }
  }
}
```

Los métodos genéricos son similares, ya que permiten crear fragmentos de código independientes del tipo de dato con el que trabajan (muy útil, por ejemplo, para algoritmos de búsqueda u ordenación). A continuación, mostramos un ejemplo de método genérico:

```
public static T[] OrdenacionBurbuja <T> (T[] valores)
{
  ...
```

```
}
```

2.4. Gestión de excepciones

Las excepciones son situaciones imprevistas que pueden suceder cuando un programa está en ejecución. En C# podemos gestionar las excepciones para que el programa no termine bruscamente en caso de error, e intentar solucionarlo en la medida de lo posible. Las excepciones son objetos de la clase *Exception* (o clases derivadas de ésta), y cuando se produce una excepción podemos capturar el objeto y tratarlo como corresponda.

Para realizar la gestión de excepciones, disponemos de *try*/*catch*/*finally*:

- ***try***: incluye el código que puede generar alguna excepción.

- ***catch***: permite capturar el objeto de la excepción. Debido a que se pueden generar excepciones de diferentes clases, podemos tener múltiples bloques *catch*.

- ***finally***: incluye código que se ejecuta con independencia de si se ha producido una excepción. Es un bloque opcional, que se suele utilizar si es necesario liberar algún recurso, por ejemplo, una conexión a una base de datos o cerrar un canal de lectura o escritura.

En el siguiente ejemplo de código, intentamos acceder fuera de los límites de un *array*, y se produce una excepción que capturamos:

```
int[] numeros = new int[2];
try
{
  numeros[0] = 10;
  numeros[1] = 15;
  numeros[2] = 33;
  Console.WriteLine(numeros[3]);
}
catch (IndexOutOfRangeException)
{
  Console.WriteLine("Índice fuera de rango");
}
catch (Exception ex)
{
  Console.WriteLine("Error desconocido: " + ex.Message);
}
finally
{
  Console.WriteLine("Aquí haríamos la limpieza.");
```

```
}
```

Cuando se produce una excepción, se revisan los bloques *catch* de modo secuencial hasta que se encuentra uno que gestione el tipo de excepción que se ha producido. Si no se encuentra ninguno, la excepción se propaga, es decir, se cancela la ejecución del método actual y se devuelve el control del programa al método que lo llamó. Si este método tampoco gestiona el error, se vuelve a propagar, y así de forma recursiva hasta que se llega al método principal. Si el método principal no gestiona el error, el programa se aborta y provoca una excepción de sistema. Una vez gestionada una excepción en un bloque *catch*, continúa la ejecución del programa en la línea siguiente al bloque *try*/*catch* en el que se gestionó.

Hay que tener en cuenta que el primer bloque *catch* que coincida con la excepción producida es el que la gestiona. Los demás *catch* no serán evaluados. Por esto, es importante capturar las excepciones de más específicas a las más genéricas, y por último *Exception*, que es la primera en la jerarquía.

A modo de ejemplo, si un *catch* gestiona la excepción *IOException* y otro gestiona *FileNotFoundException*, deberemos colocar el segundo antes que el primero para que se ejecute si se produce una excepción de tipo *FileNotFoundException*, ya que una excepción de este tipo es, a la vez, del tipo *IOException* por herencia. En caso de que tengamos un bloque *catch* sin parámetros, lo deberemos colocar en último lugar, porque se ejecuta independientemente de la excepción que se produzca.

2.4.1. Excepciones definidas por el usuario

El .NET Framework define una jerarquía de clases de excepción que parten de la clase *Exception* como raíz. Todas las clases que heredan de *Exception* son tratadas como excepciones (y por tanto se pueden utilizar en un *try*/*catch*).

Para crear una excepción personalizada, debemos crear una clase derivada de *ApplicationException*, y sobrescribir los métodos que correspondan.

Por último, nos falta saber cómo provocar una excepción. Esto es útil cuando el programa se encuentra una situación que le impide continuar, por ejemplo, el usuario no ha indicado algún valor en un formulario. En este caso, podemos crear una excepción especial llamada por ejemplo *FieldEmptyException*, y lanzarla cuando nos falte algún valor. La interfaz grafica capturará este error y avisará al usuario de que debe rellenar ese dato.

Para lanzar una excepción, utilizamos la instrucción *throw*, y creamos un objeto del tipo de la excepción que queramos provocar, por ejemplo:

```
throw new FieldEmptyException ("dni");
```

Como se puede ver en el ejemplo anterior, los objetos excepción pueden tener parámetros; en este caso, pasamos el nombre del campo que está vacío. Dependiendo del tamaño de la aplicación, se puede decidir si implementar excepciones personalizadas o no. Si la aplicación es pequeña, se puede utilizar directamente la clase *Exception* o *ApplicationException*:

```
throw new Exception("El campo nombre no puede estar vacio");
throw new ApplicationException("El campo nombre no puede estar vacio");
```

3. .NET Framework

En este apartado se ofrece una breve introducción a algunas de las clases básicas del .NET Framework, las clases de colección, y las de entrada/salida.

3.1. Clases básicas

Las clases del *namespace* System ofrecen funcionalidades básicas. En este subapartado veremos algunas de las más importantes.

3.1.1. *System.Object*

La jerarquía de clases de .NET comienza en la clase *Object*, es decir, todas las clases y demás elementos (interfaces, enumeraciones, estructuras, etc.) son, por herencia directa o indirecta, subclases de *Object*. Por lo tanto, todas las clases o elementos heredan y pueden sobrescribir los métodos de la clase *Object* para adecuarlos a sus necesidades.

Algunos de sus métodos son:

- ***Equals***: compara dos objetos y devuelve un booleano que indica si son iguales, o no.
- ***GetHashCode***: devuelve un número de *hash* que se utiliza para almacenar el objeto en tablas de *hash* (por ejemplo, la colección *Hashtable*). Idealmente, el número debe ser diferente para instancias que representan objetos diferentes.
- ***GetType***: devuelve el tipo de dato de la instancia actual.
- ***ToString***: devuelve una representación textual de la instancia actual.

Tablas de *hash*

Una tabla de *hash* es una tabla en la que cada elemento está identificado por una clave. Los elementos se insertan y se recuperan utilizando la clave correspondiente como referencia.

3.1.2. *System.Convert*

La clase *Convert* contiene una serie de métodos estáticos muy útiles que permiten convertir entre diferentes tipos de datos. Existe un método de conversión para cada tipo de dato básico: *ToInt32*, *ToDouble*, *ToChar*, *ToString*, etc. Por ejemplo, podemos convertir un *double* a *Int32* así:

```
double d = 4.7;
int i = Convert.ToInt32(d);
```

3.1.3. *System.Math*

Contiene métodos estáticos para realizar operaciones matemáticas como:

Operaciones de la clase *System.Math*	
Abs	Devuelve el valor absoluto de un número
Cos	Devuelve el coseno de un ángulo
Exp	Devuelve el elevado a una potencia
Log	Devuelve el logaritmo de un número
Pow	Devuelve la potencia de un número
Round	Devuelve un número redondeado
Sin	Devuelve el seno de un ángulo
Sqrt	Devuelve la raíz cuadrada de un número
Tan	Devuelve la tangente de un ángulo

La clase *Math* también incluye las constantes e y π.

3.1.4. *System.Random*

La clase *Random* permite generar números aleatorios. En realidad, los números generados simulan aleatoriedad a partir de un número inicial llamado semilla[7]. El constructor de la clase permite especificar un *seed* concreto:

(7)Del inglés, *seed*.

```
Random r = new Random(45);
```

Si siempre escogemos el mismo *seed*, obtendremos la misma secuencia de números aleatorios. Para aumentar la aleatoriedad, el constructor por defecto de la clase escoge un *seed* relacionado con la hora del procesador.

Una vez creada una instancia de la clase *Random*, podemos obtener números aleatorios utilizando el método *Next*. Por ejemplo, la siguiente instrucción devuelve un entero entre el 0 y el 10:

```
int i = r.Next(0, 10);
```

3.1.5. *System.String*

String es una clase que nos permite trabajar con cadenas de caracteres. *String* es un tipo especial, ya que se comporta como un tipo valor (no es necesario utilizar la palabra clave *new* para definir una variable de tipo cadena), aunque en realidad es un tipo referencia. Este tipo lo podemos escribir indistintamente como *string* o *String*.

Si delante de la cadena de caracteres ponemos el carácter @, podemos evitar los caracteres de escape y escribir caracteres como \ o salto de línea. Resulta muy útil para escribir rutas de directorios:

```
string s1 = "Hola esto es un string"
string s2 = @"c:\test\prueba.cs"
```

La clase *String* tiene multitud de métodos útiles:

Métodos de la clase *System.String*	
CompareTo	Compara dos cadenas alfanuméricamente.
IndexOf	Devuelve la posición de una subcadena.
Replace	Reemplaza una subcadena por otra.
Substring	Devuelve una cierta subcadena.
ToLower	Devuelve la misma cadena pasada a minúsculas.
ToUpper	Devuelve la misma cadena pasada a mayúsculas.
Trim	Elimina los espacios al inicio y final de la cadena.

3.1.6. *System.DateTime*

DateTime permite almacenar una fecha y hora. Podemos acceder a estos datos mediante propiedades como *Year*, *Month*, *Day*, *Hour*, *Minute*, *Second*, y podemos obtener la fecha/hora actual mediante la propiedad *Now*.

Por otro lado, contiene métodos para añadir unidades de tiempo al valor actual (*AddDays*, *AddMoths*, etc.), así como diversos métodos que permiten convertir un *DateTime* en otros tipos y viceversa. Por ejemplo, el método *Parse* permite convertir un *string* con una fecha, a un tipo *DateTime*.

El siguiente ejemplo nos indica si el año actual es bisiesto o no:

```
DateTime now = DateTime.Now;
int year = now.Year;
if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
  Console.WriteLine("El año actual es bisiesto");
```

3.1.7. *System.Array*

La clase *Array* contiene una serie de propiedades que heredan todos los *arrays*, entre ellas:

- ***Length***: devuelve el número total de posiciones del *array*.

- ***Rank***: devuelve el número de dimensiones del *array*.
- ***Clear***: inicializa las posiciones indicadas del *array*.
- ***Copy***: permite copiar partes de un *array* a otro.
- ***IndexOf***: busca un elemento en el *array*, y devuelve su posición.
- ***Sort***: ordena los elementos de un *array* unidimensional. Para poder realizar la ordenación, es necesario que los elementos del *array* implementen la interfaz *IComparable*, o proporcionar una instancia de una clase que implemente la interfaz *IComparer*, que permita comparar dos elementos del tipo del *array* entre sí.

3.1.8. *System.Environment*

La clase *Environment* permite consultar diferentes características del entorno en el cual se ejecuta la aplicación. Entre ellos, destacamos los siguientes:

- ***CurrentDirectory***: obtiene la ruta de acceso completa del directorio en el que se ha iniciado la aplicación.
- ***MachineName***: obtiene el nombre del equipo en el que se está ejecutando la aplicación.
- ***GetEnvironmentVariable***: permite consultar variables de entorno. Veamos un ejemplo para consultar el valor de la variable *PATH*:

```
string path = Environment.GetEnvironmentVariable("PATH");
```

3.2. Colecciones de datos

Los *arrays* permiten almacenar eficientemente objetos, y acceder a ellos por la posición en la que se encuentran. No obstante, los *arrays* tienen la limitación de que todos sus elementos han de ser del mismo tipo, y además, es necesario indicar la longitud exacta del *array* en el momento de su creación.

Para suplir las limitaciones de los *arrays*, en el *namespace System.Collections* disponemos de un conjunto de clases de colección, mediante los cuales podremos hacer uso de listas enlazadas, pilas, colas, tablas *hash*, etc. Algunos de estos tipos son:

- ***ArrayList***: permite almacenar objetos, y podemos acceder a ellos por su posición dentro de la estructura. Es similar a un *array*, pero con la diferencia de que *ArrayList* es una estructura dinámica, que va solicitando o liberando memoria según sea necesario.

- ***Queue***: representa una cola (estructura de datos de tipo *FIFO*).

- ***Stack***: representa una pila (estructura de datos de tipo *LIFO*).

- ***Hashtable***: representa una tabla *hash* o diccionario, en la que los elementos se identifican con una clave.

También podemos utilizar colecciones genéricas donde, al crearlas, debemos indicar el tipo de datos que van a almacenar. La siguiente tabla muestra algunas de las colecciones genéricas, y sus clases de colección equivalentes:

Colección genérica	Colección equivalente
List<T>	ArrayList
Dictionary<K,V>	Hastable
Queue<T>	Queue
Stack<T>	Stack

3.3. Entrada/Salida

El *namespace System.IO* contiene toda la funcionalidad de entrada/salida. La entrada/salida tiene lugar mediante *streams*, que son flujos de lectura o escritura sobre un cierto medio de almacenamiento, como por ejemplo la memoria, el teclado, la pantalla, o los ficheros de disco.

Streams

Gracias a la abstracción ofrecida por los *streams*, podemos acceder a diferentes tipos de dispositivos de entrada/salida haciendo uso de unas clases muy similares.

Los flujos de datos están representados por la clase *Stream*, que permite realizar diferentes operaciones:

Operaciones de la clase *Stream*	
length	Devuelve la longitud total del flujo.
position	Devuelve la posición actual en el flujo.
close	Cierra el flujo.
read	Lee una secuencia de bytes del flujo.
seek	Cambia la posición actual en el flujo.
write	Escribe una secuencia de bytes en el flujo.

La clase *Stream* es abstracta; por lo tanto, no se puede instanciar directamente, pero existen diversas subclases que sí que se pueden utilizar:

Subclases de la clase *Stream*	
FileStream	Flujo asociado a un fichero

Subclases de la clase *Stream*	
MemoryStream	Flujo en memoria
CryptoStream	Flujo de datos encriptados
NetworkStream	Flujo asociado a una conexión de red
GZipStream	Flujo de datos comprimidos

Para leer/escribir tipos de datos en un *stream*, se suele utilizar un lector o escritor[8] como los siguientes:

[8]Del inglés, *reader* y *writer*.

Lectores y escritores de un *stream*	
StreamReader	Permite leer caracteres.
StreamWriter	Permite escribir caracteres.
StringReader	Permite leer cadenas de caracteres.
StringWriter	Permite escribir cadenas de caracteres.
BinaryReader	Permite leer tipos de datos primitivos.
BinaryWriter	Permite escribir tipos de datos primitivos.

Generalmente, para leer y escribir ficheros se utilizan las clases *StreamReader* y *StreamWriter*, que pueden inicializarse directamente especificando el nombre del fichero que se quiere abrir, sin necesidad de primero crear un objeto de tipo *FileStream*. Además, estas dos clases contienen los métodos *ReadLine* y *WriteLine*, que permiten leer o escribir líneas enteras del fichero respectivamente. En el siguiente ejemplo, vemos un método que lee un fichero y lo muestra por pantalla:

```
public void LeerFichero (string file)
{
  StreamReader sr = new StreamReader (file);
  string s = sr.ReadLine ();
  while (s!=null)
  {
    Console.WriteLine (s);
    s = sr.ReadLine ();
  }
  sr.Close();
}
```

Al final de una operación con *streams* es importante cerrar el objeto lector o escritor, de modo que también se cierre el flujo de datos. En caso de no hacerlo, dejaríamos el recurso bloqueado y el resto de aplicaciones no podrían acceder a éste.

4. ADO.NET

Este apartado trata principalmente sobre ADO.NET desde el punto de vista de su arquitectura y las clases principales que lo componen, y a continuación se ofrece una introducción a LINQ.

4.1. Una introducción a ADO.NET

ADO.NET es la API de acceso a fuentes de datos de .NET. Pero antes de entrar en detalle, hagamos un breve repaso a las tecnologías de acceso a datos que ha habido en el entorno de Microsoft.

Inicialmente, las primeras bibliotecas de desarrollo para el acceso a datos eran específicas para cada tipo de base de datos concreta, por lo que no eran interoperables entre sí. En 1989, diversas empresas de software (Oracle, Informix, Ingres, DEC y otras) formaron el SQL Access Group con el objetivo de definir y promover estándares para la interoperabilidad entre bases de datos, y publicaron la especificación SQL CLI, donde se definen las interfaces a utilizar para ejecutar sentencias SQL desde otros lenguajes de programación.

En 1992, Microsoft lanzó Open Database Connectivity 1.0 (**ODBC**), basado en SQL CLI. El problema es que ODBC era complejo, lento y no basado en el modelo de componentes COM.

Microsoft desarrolló varias tecnologías sobre ODBC:

- **Data Access Object** (**DAO**) fue desarrollada para acceder a Microsoft Access. DAO era muy eficiente para bases de datos locales, pero no estaba preparada para aplicaciones con múltiples usuarios simultáneos.
- **Remote Data Objects** (**RDO**) apareció posteriormente. Su ventaja es que permite consultas complejas y accesos simultáneos.

¡Atención!

No confundir la API DAO de Microsoft con el patrón de diseño Data Access Object.

En 1996, apareció la nueva tecnología Object Linking and Embedding DataBase (**OLEDB**), que es la sucesora de ODBC. Esta tecnología tiene múltiples ventajas, como un rendimiento mucho mejor y que está basada en COM. Por encima de OLE DB Microsoft desarrolló **ActiveX Data Objects** (**ADO**), con el objetivo de sustituir a DAO y RDO.

En el 2002, apareció **ADO.NET** como tecnología de acceso a fuentes de datos de .NET, que hereda las mejores características de ADO, y proporciona nuevas funcionalidades como, por ejemplo, la posibilidad de trabajar tanto de forma conectada como desconectada de las bases de datos.

Las **características principales de ADO.NET** son:

a) Permite trabajar tanto de forma conectada como desconectada del origen de datos. Una de las ventajas del acceso desconectado es que se consigue una mayor escalabilidad debido a que las conexiones de los usuarios no se mantienen por periodos largos, con lo que se puede permitir el acceso de más usuarios.

b) Tiene una fuerte integración con XML, lo que lo hace más fácil de compartir, interpretar y trabajar con él.

c) Es independiente del lenguaje de programación utilizado.

d) No sólo permite acceder a bases de datos, sino también a otras fuentes de datos como hojas de cálculo, XML, texto, etc.

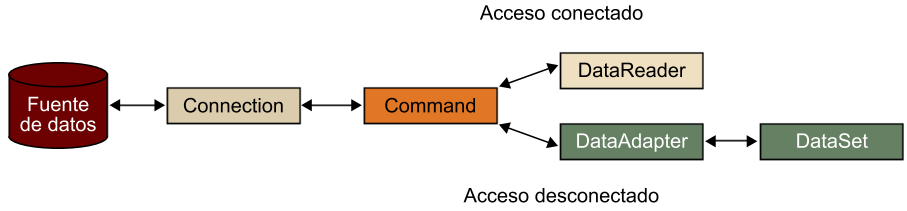
Como se ha comentado, ADO.NET permite trabajar de dos formas:

- **Acceso conectado**: requiere una conexión permanente con la base de datos (hasta la llegada de ADO.NET, esto era lo más habitual).

- **Acceso desconectado**: un subconjunto de los datos de la fuente de datos se copian en un *dataset* y, si luego se producen cambios en el *dataset*, éstos se propagan a la fuente de datos.

La figura siguiente muestra las clases de .NET que intervienen en cada caso:

Figura 2. Modos de acceso de ADO.NET



Las clases principales de ADO.NET son las siguientes:

- ***Connection***: realiza la conexión con una fuente de datos.

- ***Command***: permite hacer consultas u operaciones de modificación contra una fuente de datos.

- ***DataReader***: permite acceder a los resultados de una consulta realizada contra una fuente de datos.

- ***DataAdapter***: lee los datos y los carga en el *dataset*. Si ahí se producen cambios, el data adapter será el encargado de sincronizar todos los cambios en la fuente de datos.

- ***DataSet***: permite almacenar y manipular datos en memoria.

Para establecer una **conexión con la fuente de datos**, en primer lugar hemos de seleccionar el proveedor de datos adecuado, y a continuación tener en cuenta las cadenas de conexión.

4.1.1. Proveedores de datos

Un proveedor de datos consta de un conjunto de clases que permiten el acceso y la comunicación con las fuentes de datos. Dentro del .NET Framework, se incluyen los siguientes proveedores de datos:

Otros proveedores

Existen muchos otros proveedores de datos que no vienen integrados con el .NET Framework, y los hay tanto gratuitos como comerciales.

- **Proveedor de datos para ODBC**: proveedor genérico que permite acceder a cualquier fuente de datos mediante el driver ODBC correspondiente (necesario tenerlo instalado previamente).

- **Proveedor de datos para OLEDB**: proveedor genérico que permite acceder a cualquier fuente de datos mediante el driver OLE DB (necesario tenerlo instalado previamente). En comparación con el proveedor ODBC, éste es más recomendable ya que, entre otros motivos, es más rápido.

- **Proveedor de datos para SQL Server**: proveedor específico para SQL Server 7.0 o posterior. La comunicación con SQL Server tiene lugar sin capas intermedias (ni OLEDB ni ODBC), con lo que el rendimiento es aún mejor.

- **Proveedor de datos para Oracle**: es un proveedor de datos específico para Oracle 8.1.7 o posterior, que también ofrece el mayor rendimiento posible, al no hacer uso de ninguna capa intermedia (ni OLE DB ni ODBC).

Todos los proveedores de datos han de ofrecer las siguientes clases:

Clase	Descripción
xxxConnection	Permite establecer una conexión a un tipo de fuente de datos.
xxxCommand	Permite ejecutar un comando SQL en una fuente de datos.
xxxDataReader	Permite leer un conjunto de datos de la fuente de datos
xxxDataAdapter	Permite cargar y actualizar datos en un objeto DataSet

4.1.2. Cadenas de conexión

La cadena de conexión es una cadena de caracteres que identifica las características de la conexión con una cierta fuente de datos. La cadena de conexión incluye la localización del servidor, el nombre de la base de datos, el usuario, el *password*, u otras opciones específicas del proveedor de datos utilizado. La siguiente tabla muestra los parámetros más comunes:

Parámetro	Descripción
Data Source	Nombre del proveedor de la conexión
Initial Catalog	Nombre de la base de datos
User ID	Usuario de acceso a la fuente de datos
Password	*Password* de acceso a la fuente de datos

A modo de ejemplo, el siguiente código se conecta a una base de datos SQL Server con el usuario sa, *password* xxx, y accede a la base de datos Northwind del servidor miBBDD:

```
SqlConnection myConnection = new SqlConnection();
myConnection.ConnectionString = "Data Source=miBBDD;" +
                                "Initial Catalog=Northwind;" +
                                "User ID=sa;Password=xxx"
myConnection.Open();
```

Acceso conectado

El acceso conectado consiste en abrir una conexión con la fuente de datos, ejecutar una serie de sentencias SQL y cerrar la conexión. La clase Command ofrece los siguientes métodos para ejecutar las sentencias:

Método	Descripción
ExecuteScalar	Ejecuta una sentencia que devuelve un único valor.
ExecuteReader	Ejecuta una sentencia que devuelve un conjunto de filas.
ExecuteNonQuery	Ejecuta una sentencia que modifica la estructura o los datos de la base de datos. Devuelve el número de filas afectadas.
ExecuteXmlReader	Ejecuta una sentencia que devuelve un resultado XML.

Sentencias que no devuelven valores

Utilizaremos el método *ExecuteNonQuery* para ejecutar sentencias SQL que no devuelven valores. El siguiente ejemplo crea una tabla llamada A:

```
SqlCommand cmd = new SqlCommand (
  "CREATE TABLE A (A INT, PRIMARY KEY (A))", conn);
cmd.ExecuteNonQuery ();
```

En el caso de las sentencias SQL de manipulación (DML), el método *ExecuteNonQuery* devuelve el número de filas afectadas. El siguiente ejemplo modifica el precio de los productos y obtiene el número de filas afectadas:

```
SqlCommand cmd = new SqlCommand (
```

```
    "UPDATE PRODUCTOS SET precio = precio + 10", conn);
int numFilas = cmd.ExecuteNonQuery ();
```

Sentencias que devuelven un único valor

El método *ExecuteScalar* lo utilizaremos cuando la sentencia devuelva un único valor. El siguiente ejemplo devuelve el número de filas de una tabla:

```
SqlCommand cmd = new SqlCommand (
  "SELECT COUNT(*) FROM PRODUCTOS", conn);
int numFilas = (int)(cmd.ExecuteScalar ());
```

Sentencias que devuelven un conjunto de valores

En general, la ejecución de una sentencia *SELECT* o un procedimiento almacenado devuelve un conjunto de datos. Para ejecutar este tipo de sentencias utilizamos el método *ExecuteReader* de la clase *Command*, que devuelve un objeto *DataReader* que nos permite consultar las filas obtenidas.

Un *DataReader* es un cursor que permite iterar hacia delante sobre un conjunto de filas (no se pueden recuperar elementos anteriores), resultado de la ejecución de una sentencia SQL o procedimiento almacenado. Para poder utilizar un *DataReader* es necesario mantener abierta la conexión con la base de datos; no podemos cerrarla y después recorrer el *DataReader*. Por eso, se denomina acceso a datos conectado.

En el siguiente ejemplo, realizamos una consulta y obtenemos los resultados:

```
SqlCommand cmd = new SqlCommand(
  "SELECT id, descr FROM usuarios", conn);
SqlDataReader reader = cmd.ExecuteReader();
while (reader.Read())
{
  int id = reader.GetInt32(0);
  String descr = reader.GetString(1);
}
reader.Close();
conn.Close();
```

Comprobación de valores *null*

Podemos comprobar cuándo el valor de una columna es nulo (es decir, es un valor *null*) con el método *IsDbNull* del objeto *DataReader*, como podemos ver en el siguiente ejemplo:

Tipos anulables

Los tipos anulables son una extensión de los tipos valor de .NET, y permiten que el tipo de datos pueda contener también el valor *null*. Para indicarlo, añadimos un ? junto al tipo de datos.

```
if (!reader.IsDBNull (3)) // Si la tercera columna no es null
{
```

```
  int i = dr.GetInt32(3);
}
```

A partir de la versión 2.0 del .NET Framework, gracias a los tipos anulables, podemos ejecutar el método *GetXXX* sin comprobaciones adicionales:

```
int? i = dr.GetInt32(3);
```

Procedimientos almacenados

Los procedimientos almacenados pueden o no devolver resultados. Si devuelven resultados, los ejecutaremos mediante el método *ExecuteReader* de la clase *Command*, mientras que si no devuelven resultados los ejecutaremos mediante el método *ExecuteNonQuery*.

Para ejecutar un procedimiento almacenado hay que crear un *SqlCommand*, al que se le indica el nombre y los parámetros del procedimiento:

```
SqlCommand cmd = new SqlCommand ("ActualizarPrecio", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add ("@productoID", 4);
cmd.Parameters.Add ("@nuevoPrecio", 50);

int modified = cmd.ExecuteNonQuery ();
Console.WriteLine ("Se han modificado " + modified + " filas");
```

Transacciones

Transacciones

Conjunto de sentencias relacionadas que se deben ejecutar todas o, si no, no se puede ejecutar ninguna. Se dice que una transacción hace *commit* o *rollback*, respectivamente.

ADO.NET permite gestionar transacciones de forma sencilla.

En el siguiente ejemplo, ejecutamos dos sentencias SQL dentro de una transacción:

```
String connString = "Data Source=miBBDD;" +
  "Initial Catalog=Northwind;" +
  "User ID=sa;Password=xxx";
SqlConnection conn = new SqlConnection(connString);
conn.Open();
SqlTransaction tran = conn.BeginTransaction();
SqlCommand command1 = new SqlCommand(
  "DELETE FROM User WHERE Id = 100", conn, tran);
SqlCommand command2 = new SqlCommand(
  "DELETE FROM User WHERE Id = 200", conn, tran);

try
{
  command1.ExecuteNonQuery();
```

```
    command2.ExecuteNonQuery();
    tran.Commit();
  }
  catch (SqlException)
  {
    tran.Rollback();
  }
  finally
  {
    conn.Close();
  }
```

Acceso desconectado

En este subapartado se explican los diversos tipos de acceso desconectado que pueden darse.

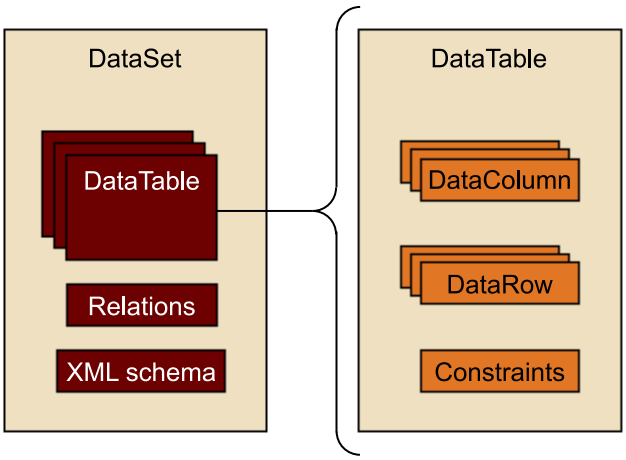
Datasets no tipados

Un *DataSet* es una copia en memoria de una parte de una base de datos. Constituye una vista "desconectada" de los datos, es decir, existe en memoria sin una conexión activa a una base de datos que contenga la correspondiente tabla o vista. El objeto *DataSet* guarda los datos y su estructura en XML, lo que permite por ejemplo enviar o recibir un XML por medio de HTTP.

Un *DataSet* está compuesto por tablas (*DataTables*) y una lista de relaciones entre éstas (*Relations*). Además, el *DataSet* mantiene un *schema* XML con la estructura de los datos. Un *DataTable*, a su vez, está formado por columnas (*DataColumn*) y contiene una serie de filas de datos (*DataRow*). También puede tener definidas una serie de restricciones (*Constraints*), como *Primary key*, *Foreign key*, *Unique* o *Not null*.

Las filas constituyen los datos que se almacenan en el *DataTable*. De cada *DataRow* se guardan dos copias, una de la versión inicial (la que se recupera de la base de datos), y otra de la versión actual, para identificar los cambios realizados en el *DataSet*.

Figura 3. Estructura de un *DataSet*



El siguiente ejemplo crea un objeto *DataTable* que representa la tabla PRODUCTOS de una fuente de datos, y lo añade al *DataSet* ALMACEN:

```
DataSet ds = new DataSet("ALMACEN");
DataTable dt = ds.Tables.Add("PRODUCTOS");
dt.Columns.Add("id", typeof (Int32));
dt.Columns.Add("nombre", typeof (String));
dt.Columns.Add("precio", typeof (Double));
```

Para recuperar tablas o columnas de las colecciones *Tables* y *Columns*, podemos acceder por posición o por el nombre del elemento. En el siguiente ejemplo, recuperamos la columna "id" de la tabla PRODUCTOS:

```
// acceso por posición
DataColumn dc = ds.Tables[0].Columns[0];

// acceso por nombre
DataColumn dc = ds.Tables["PRODUCTOS"].Columns["id"];
```

Datasets tipados

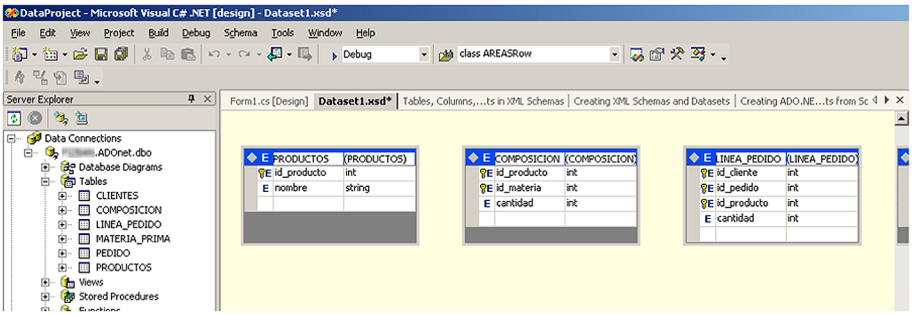
Un *dataset* tipado es una clase derivada de *DataSet*, y que tiene un esquema de tablas y relaciones predefinido. Además, proporciona métodos y propiedades para acceder a las tablas, filas y columnas del *DataSet*, utilizando los nombres de éstas, permitiendo un código mucho más sencillo y legible.

El .NET Framework SDK proporciona la herramienta xsd.exe que, dado un *schema* XML (XSD) que define la estructura del *DataSet*, crea la clase *DataSet* que corresponde a la estructura indicada. Esta clase se puede compilar por separado, o añadir directamente con el resto de clases de la aplicación.

Para facilitar la tarea de creación de *datasets* con tipo, Visual Studio proporciona herramientas de diseño visual a tal efecto. Para ello, hay que añadir un nuevo elemento al proyecto de tipo *DataSet*, con lo cual aparece el diseñador de *datasets*, donde podemos definir su estructura en forma de XSD.

Otra opción más rápida es crear la estructura de un *DataSet* a partir de una base de datos existente. Para ello, en la ventana del explorador de servidores, abrimos la lista de tablas de la base de datos, y arrastramos las tablas que queramos añadir al *DataSet*, directamente al diseñador de *DataSet*.

Figura 4. Creación de un *DataSet* tipado



Una vez creada la estructura de un *dataset*, podemos añadirle nuevas filas de datos, hacer modificaciones o eliminaciones. Una vez se han producido estos cambios, podemos hacerlos permanentes o rechazarlos con los métodos *AcceptChanges* y *RejectChanges*. Concretamente, si ejecutamos *AcceptChanges* sobre una fila, todos los cambios pasan a ser datos permanentes. En cambio, si ejecutamos *RejectChanges*, los cambios se rechazarán y la fila vuelve al estado anterior a todos esos cambios.

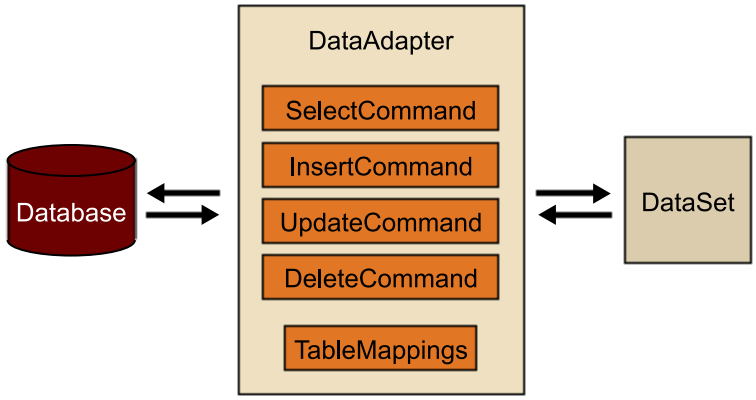
La clase *DataAdapter*

La clase *DataAdapter* actúa de intermediaria entre la fuente de datos y un *DataSet*. El *DataAdapter* es el encargado de conectarse a la fuente de datos, ejecutar la consulta SQL deseada, y llenar el *DataSet* con los datos obtenidos. También es el encargado de actualizar la fuente de datos con los cambios realizados en el *DataSet*, una vez se ha acabado de trabajar con él.

Esta clase forma parte del proveedor de datos que se utiliza, por ejemplo para el proveedor SQLServer.NET, será *SqlDataAdapter*, y para el de OLEDB será *OleDbDataAdapter*.

Como vemos en el siguiente diagrama, un *DataAdapter* contiene una serie de objetos *Command* que representan las sentencias de consulta (*SelectCommand*), inserción de nuevos datos (*InsertCommand*), modificación de datos (*UpdateCommand*), y borrado de datos (*DeleteCommand*) respectivamente. Además, contiene una propiedad *TableMappings* que relaciona las tablas del *DataSet* con las tablas homólogas de la fuente de datos.

Figura 5. La clase *DataAdapter*



La clase *DataAdapter* proporciona el método *Fill*, que permite llenar un objeto *DataSet* con los resultados de la consulta SQL representada por el objeto *SelectCommand*. El siguiente ejemplo muestra cómo crear un objeto *DataAdapter*, y cómo rellenar un *dataset* con el método *Fill*:

```
SqlConnection conn = new SqlConnection(strConn);

SqlDataAdapter da = new SqlDataAdapter (
  "SELECT id, nombre FROM CLIENTES", conn);

DataSet ds = new DataSet ();

da.Fill(ds, "CLIENTES");
```

Como se muestra en el ejemplo, no hay que llamar explícitamente a los métodos *Open* y *Close* de la conexión. El método *Fill* comprueba si la conexión está o no abierta, y si no lo está la abre, y la cierra al acabar.

Una vez se hayan realizado todos los cambios pertinentes, utilizamos de nuevo la clase *DataAdapter* para actualizar los datos de la fuente de datos con las modificaciones realizadas en el *DataSet* con el método *Update*, que utiliza las sentencias SQL representadas por las propiedades *InsertCommand*, *UpdateCommand* y *DeleteCommand*, para insertar las filas nuevas, modificar las ya existentes, y borrar las eliminadas respectivamente.

- **Problemas de concurrencia**
  El uso de *DataSets* en una base de datos a la que acceden múltiples usuarios añade el problema de la gestión de la concurrencia. Mientras un usuario está trabajando con los datos en una copia local, se puede producir una actualización de éstos en la fuente de datos real, pudiendo invalidar los datos que leímos, cuando todavía estamos trabajando con ellos. En general, existen varias técnicas de gestión de concurrencia:

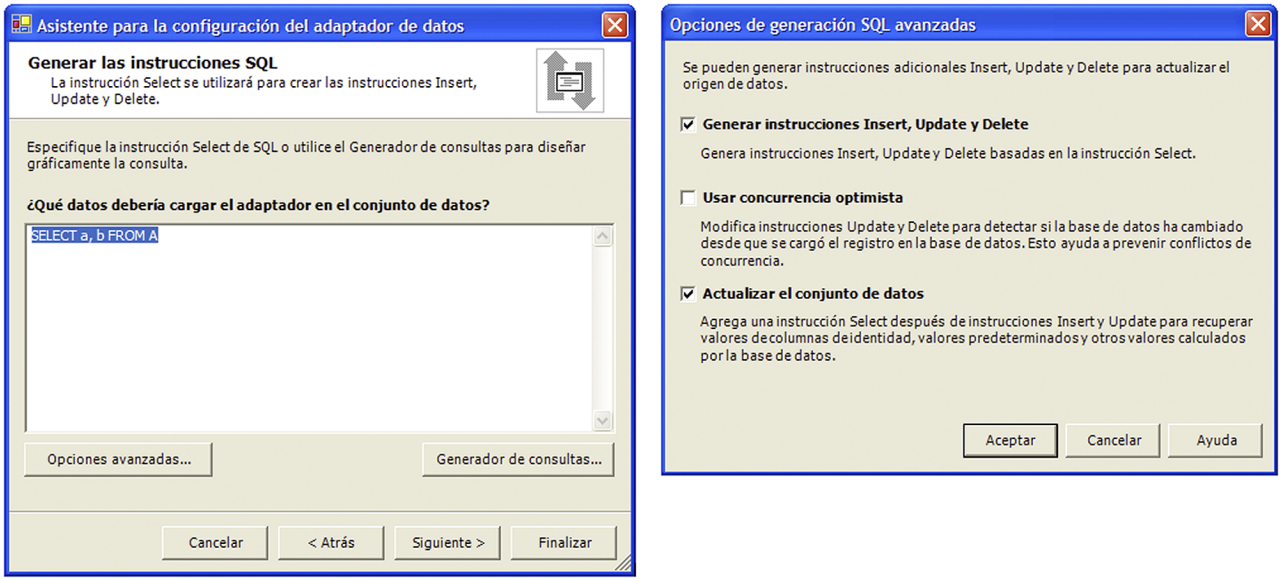
– **Concurrencia pesimista**: cuando una fila es leída, ésta queda bloqueada para su lectura para cualquier otro que la demande, hasta que aquél que la posee la libere.

– **Concurrencia optimista**: las filas están disponibles para su lectura en todo momento; éstas pueden ser leídas por distintos usuarios al mismo tiempo. Cuando alguno intenta modificar una fila que ya fue modificada, se produce un error y no se produce la modificación.

– ***Last win***: esta técnica implica que no existe control alguno; simplemente, la última actualización es la que permanece.

Visual Studio sólo permite escoger entre concurrencia optimista o *last win* al crear un objeto *DataAdapter* con el asistente de configuración (ved la figura siguiente). En caso de no seleccionar concurrencia optimista, se aplica por defecto concurrencia de tipo *last win*.

Figura 6. Asistente del *DataAdapter*



La clase *DataView*

Un objeto *DataView* es similar a una vista de una base de datos, donde podemos hacer filtrados u ordenación de los datos. En el siguiente ejemplo, creamos una vista ordenada por nombre, y con un filtro de precio:

```
DataView dv = new DataView (ds.Tables("PRODUCTOS"));
dv.Sort = "nombre";
dv.RowFilter = "precio > 100";
```

4.2. LINQ

LINQ añade capacidades de consulta y modificación de datos a los lenguajes C# y Visual Basic .NET. LINQ no sólo permite trabajar con bases de datos, sino que está diseñado para trabajar con cualquier tipo de origen de datos, como listas de objetos, archivos XML, etc.

La diferencia principal entre utilizar ADO.NET o LINQ, para consultar una fuente de datos, es que con LINQ las consultas se comprueban de forma sintáctica en tiempo de compilación.

Disponemos de las siguientes implementaciones de LINQ:

1) LINQ to Objects: permite realizar consultas sobre colecciones de objetos. Es el que utilizaremos en todos los ejemplos de este subapartado.

2) LINQ to XML: permite realizar consultas sobre datos en formato XML.

3) LINQ to ADO.NET: permite consultar bases de datos a través de ADO.NET. Existen tres variantes:

a) LINQ to SQL: permite hacer consultas sobre bases de datos relacionales, aunque inicialmente sólo con SQL Server.

b) LINQ to DataSet: permite realizar consultas sobre DataSets.

c) LINQ to Entities: permite realizar consultas sobre un modelo conceptual de datos. Está relacionado con ADO.NET Entity Framework, que permite trabajar contra un modelo conceptual de datos, sin preocuparse de la estructura real de la base de datos.

4.2.1. Sintaxis de LINQ

La sintaxis de LINQ es similar al lenguaje SQL. En LINQ, el resultado de una consulta es siempre un tipo enumerable (concretamente, es una instancia de *IEnumerable<T>*), con lo que siempre podremos recorrer el resultado tal y como haríamos con una enumeración cualquiera.

¡Atención!

Para trabajar con LINQ es necesario añadir la sentencia "using System.Linq", y que el proyecto esté configurado para trabajar con .NET Framework 3.5.

El siguiente ejemplo hace una consulta que devuelve los nombres de las personas del *array* personas cuyo lugar de nacimiento sea Barcelona, que son Rosa y Juan:

```
Persona[] personas = new Persona[] {
  new Persona ("Juan", "Barcelona", 20),
  new Persona ("Pedro", "Londres", 30),
  new Persona ("Maria", "Lisboa", 40),
  new Persona ("Rosa", "Barcelona", 25)};
```

```
var personas_BCN =
  from p in personas
  where (p.Ciudad == "Barcelona")
  orderby p.Edad descending, p.Nombre
  select (p);

foreach (Persona p in personas_BCN)
  Console.WriteLine(p.Nombre);
```

La palabra clave *var* se utiliza para inferir automáticamente el tipo de dato del resultado. En el ejemplo anterior, sería equivalente haber escrito *IEnumerable<Persona>* en lugar de *var*. Algunos de los operadores de LINQ son:

- ***from***: indica la fuente de información sobre la que se va a ejecutar la consulta.
- ***where***: permite especificar las restricciones o filtros a aplicar a los datos.
- ***select***: indica los campos a devolver en el resultado.
- ***orderby***: permite ordenar los resultados por uno o varios criterios de ordenación, y podemos indicar el orden como *ascending* (por defecto) o *descending*.
- ***group by***: permite agrupar los resultados según un determinado criterio. El siguiente ejemplo muestra cómo agrupar las personas por ciudad:

```
Persona[] personas = new Persona[] {
new Persona ("Juan", "Barcelona", 20),
new Persona ("Pedro", "Londres", 30),
new Persona ("Maria", "Lisboa", 40),
new Persona ("Rosa", "Barcelona", 25)};

var grupos = from p in personas
   group p by p.Ciudad;

foreach (var grupo in grupos)
{
   Console.WriteLine("Ciudad: " + grupo.Key);
   Console.WriteLine("Cont: " + grupo.Count());
   foreach (Persona persona in grupo)
   {
      Console.WriteLine(" " + persona.Nombre);
   }
}
```

Al ejecutar el ejemplo anterior, obtenemos el siguiente resultado:

```
Ciudad: Barcelona
Cont: 2
```

```
  Juan
  Rosa
Ciudad: Londres
Cont: 1
  Pedro
Ciudad: Lisboa
Cont: 1
  Maria
```

- ***join***: el operador *join* permite definir una relación entre dos clases o entidades dentro de la consulta. Su semántica es similar a la del *join* de SQL, ya que lo que hace es cruzar ambas tablas (o clases) en función de la expresión de enlace especificada. A continuación, se muestra un ejemplo en el que se utiliza la instrucción *join* para combinar las personas con las ciudades, y obtener un listado con el nombre de la persona, su ciudad, y el país:

```
Persona[] personas = new Persona[] {
   new Persona ("Juan", "Barcelona", 20),
   new Persona ("Pedro", "Londres", 30),
   new Persona ("Maria", "Lisboa", 40),
   new Persona ("Rosa", "Barcelona", 25)};

Ciudad[] ciudades = new Ciudad[] {
   new Ciudad ("Barcelona", "España"),
   new Ciudad ("Londres", "Reino Unido"),
   new Ciudad ("Lisboa", "Portugal")};

var listado =
   from p in personas
   join c in ciudades
   on p.Ciudad equals c.Nombre
   select new {p.Nombre, p.Ciudad, c.Pais};

foreach (var p in listado)
   Console.WriteLine(p.Nombre+"\t"+p.Ciudad+"\t"+p.Pais);
```

Al ejecutar el ejemplo anterior, obtenemos el siguiente resultado:

```
Juan       Barcelona        España
Pedro      Londres          Reino Unido
Maria      Lisboa           Portugal
Rosa       Barcelona        España
```

Agregados

Los agregados son métodos que devuelven el resultado de realizar una determinada operación sobre los elementos de la lista de valores sobre la que se aplica. A continuación, se indica la utilidad de cada uno de ellos:

Método	Descripción
Count	Devuelve el número de elementos de la lista.
Min	Devuelve el menor elemento.
Max	Devuelve el mayor elemento.
Sum	Devuelve la suma de los valores de la lista.
Average	devuelve una media de los valores de la lista

El siguiente ejemplo muestra la edad mínima de las personas del *array*, lo que ejecutado sobre el *array* de personas del ejemplo anterior, nos daría como resultado el valor 20:

```
var edad_minima = ( from p in personas
                    select p.Edad).Min();
Console.WriteLine(edad_minima);
```

5. Windows Forms

En este apartado, se introduce la tecnología Windows Forms, que permite crear aplicaciones de escritorio.

Windows Forms es una tecnología de Microsoft que permite desarrollar aplicaciones de escritorio de forma sencilla. Antes de su aparición, se utilizaban las MFC (Microsoft Foundation Classes), donde la programación era bastante más compleja.

Windows Forms aparece en el año 2002, y consta de un conjunto de clases que permiten desarrollar interfaces de usuario, ya sea añadiendo controles gráficos que gestionamos mediante eventos, o realizando llamadas a bajo nivel mediante las clases de System.Drawing, con las que podemos dibujar, escribir texto, procesar imágenes, etc.

El uso por parte del programador de bibliotecas gráficas más complejas, como la librería GDI+ de Win32, DirectX u OpenGL (Open Graphics Library) queda relegado sólo a aquellas situaciones que requieran funcionalidades gráficas más avanzadas como procesamiento de imágenes, animaciones, 3D, etc.

Librería GDI+

Aunque el programador directamente no utilice GDI+ (del inglés, *graphics device interface*), las llamadas a la API de WinForms internamente se acaban traduciendo a llamadas a GDI+.

Un formulario es la ventana utilizada para presentar la información, o recibir las entradas del usuario. Un formulario puede ser una simple ventana, una ventana MDI, o un diálogo. Y en todos los casos, cabe tener presente que un formulario es un objeto de una cierta clase (concretamente, cada formulario es una clase derivada de *System.Windows.Forms.Form*), con lo que presentará un conjunto de propiedades, métodos y eventos:

Ventana MDI

MDI, del inglés *multiple document interface,* son aquellas ventanas que dentro pueden contener diversas ventanas. Ejemplos de aplicaciones MDI son Adobe Photoshop, o Visual Studio de Microsoft.

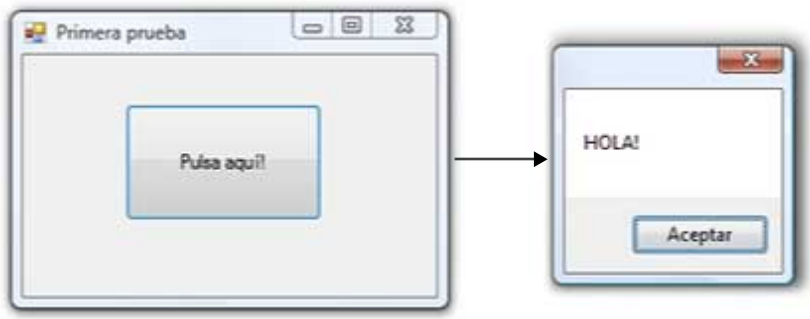
- **Propiedades**: permiten, por ejemplo, cambiar la apariencia de un formulario (su color, tamaño, posición, etc.).
- **Métodos**: exponen el comportamiento del formulario (mostrar, ocultar, cerrar, mover, etc.).
- **Eventos**: permiten interactuar con el formulario y asociar el código a ejecutar cuando se produzcan estos eventos (al cerrar el formulario, al minimizarlo, al moverlo, etc.).

Un formulario contiene internamente un conjunto de controles (botones, etiquetas, cajas de texto, etc.) que nos permitirán crear la interfaz de usuario. Así como el formulario tiene unos eventos asociados, cada uno de los controles también tendrá sus propios eventos. Para todos aquellos eventos que nos inte-

resen, implementaremos la función correspondiente, que contendrá el código a ejecutar cuando se produzca el evento (por ejemplo, cuando el usuario pulse un botón, se mostrará un mensaje por pantalla).

Para entender el código generado automáticamente por Visual Studio, crearemos una primera aplicación de tipo Windows Forms, a la que añadiremos un botón, que al pulsarlo mostrará un mensaje "HOLA!":

Figura 7. Primera aplicación WinForms



Veamos el código correspondiente:

a) Form1.cs: contiene el constructor y el código asociado a los eventos.

```
public partial class Form1 : Form
{
  public Form1()
  {
    InitializeComponent();
  }

  private void Button1_Click(object sender, EventArgs e)
  {
    MessageBox.Show("HOLA!");
  }
}
```

- Al crear el formulario con Visual Studio, se ha creado automáticamente una clase *Form1*, que deriva de *Form*.

- La clase es *partial*, lo que permite repartir su código en dos archivos: Form1.cs, donde el programador escribirá el código asociado a los evento y, el otro, Form1.Designer.cs, generado automáticamente por Visual Studio a partir del diseño que hagamos del formulario, y que el programador en principio no tendrá que editar nunca.

- El constructor llama al método *InitializeComponent*, que ha sido generado por Visual Studio, y es donde se inicializan todos los controles del formulario.

- El método *Button1_Click* será llamado cuando el usuario pulse el botón *Button1*. En ese momento, aparecerá el mensaje "HOLA!" por pantalla.

b) Form1.Designer.cs: contiene el código generado automáticamente por Visual Studio (se recomienda no modificarlo manualmente).

```
partial class Form1
{
  private IContainer components = null;

  protected override void Dispose(bool disposing)
  {
    if (disposing &amp;&amp; (components != null))
    {
      components.Dispose();
    }
    base.Dispose(disposing);
  }

  private void InitializeComponent()
  {
    this.Button1 = new Button();
    this.SuspendLayout();

    this.Button1.Location = new Point(65, 31);
    this.Button1.Name = "Button1";
    this.Button1.Size = new Size(125, 74);
    this.Button1.TabIndex = 0;
    this.Button1.Text = "Pulsa aquí!";
    this.Button1.UseVisualStyleBackColor = true;
    this.Button1.Click +=
      new EventHandler(Button1_Click);

    this.AutoScaleDimensions = new SizeF(6F, 13F);
    this.AutoScaleMode = AutoScaleMode.Font;
    this.ClientSize = new Size(263, 153);
    this.Controls.Add(this.Button1);
    this.Name = "Form1";
    this.Text = "Primera prueba";
    this.ResumeLayout(false);
  }

  private Button Button1;
```

```
}
```

- El método *Dispose* ha sido creado por Visual Studio, y contiene el código para liberar los recursos del formulario.

- El método *InitializeComponent* también ha sido creado por Visual Studio. En el código, se crea una instancia del botón, se inicializan sus propiedades, y se asigna el método *Button1_Click* al evento *Click* del botón. A continuación, se inicializan las propiedades del formulario, como por ejemplo el tamaño o título de la ventana.

- El formulario contiene una variable para cada uno de sus controles En este caso, como sólo hay un botón, contiene una única variable llamada *Button1*, que es de la clase *Button*.

5.1. Implementación de eventos

La implementación de los eventos de C# está basada en el uso de delegados, que permiten pasar funciones como parámetros a otras funciones. Concretamente, un delegado permite al programador encapsular una referencia a un método dentro de un objeto delegado. Posteriormente, el objeto delegado podrá ser pasado como parámetro a un código que se encargará de llamar al método referenciado, sin necesidad de tener que saber en tiempo de compilación cuál es el método en cuestión.

Equivalencia

En C/C++ lo equivalente a los delegados serían los punteros a funciones.

5.1.1. Delegados

Un delegado permite almacenar una referencia a cualquier método que cumpla con un cierto contrato, que viene definido por el tipo y número de parámetros, y por el tipo del valor de retorno de la función. A modo de ejemplo, el siguiente delegado permitirá referenciar cualquier función que reciba dos enteros como parámetros, y devuelva otro entero:

```
public delegate int MiDelegado (int valor1, int valor2);
```

En el siguiente ejemplo, se define y utiliza un delegado. Al ejecutar la aplicación, se mostraría por la pantalla el valor "8" correspondiente al valor de 2^3:

```
class Program
{
  public delegate int MiDelegado(int valor1, int valor2);

  public static int suma(int valor1, int valor2)
  {
    return valor1 + valor2;
  }
```

```
  public static int potencia(int bas, int exponente)
  {
    return (int)Math.Pow(bas, exponente);
  }

  public static void calculo(MiDelegado f, int a, int b)
  {
    Console.WriteLine(f(a, b));
  }

  static void Main(string[] args)
  {
    MiDelegado delegado = new MiDelegado(potencia);
    calculo(delegado, 2, 3);
  }
}
```

5.1.2. Funciones gestoras de eventos

Las funciones asociadas a los eventos se gestionan mediante delegados con el siguiente contrato:

```
delegate void NombreEventHandler (object Sender, EventArgs e);
```

- La función no retorna nada (*void*).
- El primer parámetro es el objeto que ha generado el evento.
- El segundo parámetro almacena los datos que pueden ser utilizados en la función gestora del evento. Puede tratarse de un objeto de la clase *EventArgs* o, si no, de una clase derivada que permita almacenar información adicional en función del tipo de evento.

En el siguiente ejemplo, mostramos diversas funciones gestoras de eventos:

```
private void Form2_Load(object sender, EventArgs e)
{
}

private void textBox1_KeyPress( object sender, KeyPressEventArgs e)
{
  MessageBox.Show(e.KeyChar.ToString());
}

private void textBox1_MouseClick(object sender,MouseEventArgs e)
{
  MessageBox.Show(e.X + " " + e.Y);
```

```
}
```

- La primera corresponde con el evento *Load* del formulario, que se ejecuta una única vez, justo antes de que el formulario se muestre por pantalla por primera vez. Este evento es muy utilizado, ya que es el que añade Visual Studio automáticamente por defecto al hacer una doble pulsación sobre un formulario cualquiera.

- La segunda función corresponde con la pulsación de una tecla en un control *TextBox*, donde se muestra por pantalla la tecla pulsada.

- La tercera función corresponde a la pulsación de un botón del *mouse* sobre un control *TextBox*, que muestra por pantalla la posición (X,Y) donde ha tenido lugar la pulsación.

5.2. Controles

En un formulario, podemos arrastrar múltiples elementos de la paleta de controles aunque, para ser precisos, la paleta contiene tanto contenedores como componentes y controles. Veamos, a continuación, las diferencias entre unos y otros:

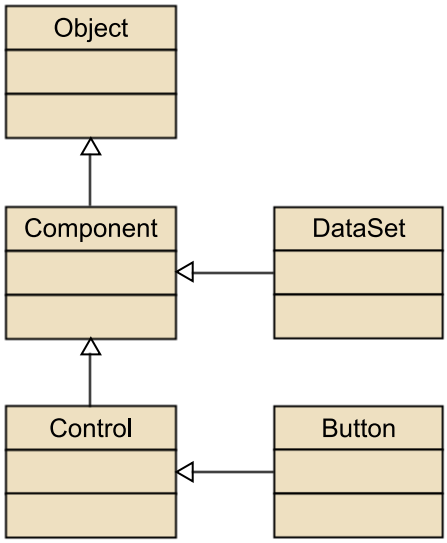
- Un contenedor es un objeto que puede contener múltiples componentes. Por citar algunos ejemplos, en la paleta de controles disponemos de los contenedores *Panel*, *GroupBox* o *TabControl*.

- Un componente es una subclase de *Component*, que ofrece una interfaz claramente definida, y que está diseñada para ser reutilizada por terceras aplicaciones, pero que no tiene por qué mostrar ninguna interfaz gráfica. Un ejemplo sería el componente *DataSet*, que permite almacenar datos, pero no muestra nada por pantalla (para ello, será necesario utilizar otros controles).

- Un control es una clase derivada de *Control*, y es un componente que además tiene interfaz gráfica. Un ejemplo sería el control *Button*, que muestra un botón.

En la siguiente figura, mostramos de forma muy simplificada la jerarquía de clases correspondiente al componente *DataSet* y al control *Button*:

Figura 8. Jerarquía simplificada de componentes y controles



.NET Framework proporciona multitud de componentes y controles que nos permiten desarrollar interfaces gráficas de forma rápida y sencilla. Además, en caso que no encuentre el control necesario, podemos implementar nuestros propios controles personalizados.

Controles adicionales

En Internet podemos encontrar multitud de controles adicionales, y los hay tanto gratuitos como comerciales.

A continuación, se repasan brevemente los elementos principales de la paleta de controles de Visual Studio: contenedores, controles, componentes y diálogos.

5.2.1. Contenedores

En la paleta de controles de Visual Studio podemos encontrar los contenedores siguientes:

- ***Panel***: permite agrupar un conjunto de controles, y tratarlos de forma simultánea. A modo de ejemplo, nos facilitaría mostrar u ocultar todo un conjunto de controles por pantalla.

- ***SplitContainer***: son dos paneles separados por una barra movible (llamada *splitter*) que el usuario puede utilizar para redimensionar el tamaño dedicado a cada uno de los paneles.

- ***GroupBox***: Es muy similar al Panel, y la diferencia principal es la apariencia que muestra por pantalla.

- ***TabControl***: permite albergar controles en diferentes grupos, y el usuario puede acceder a unos u otros mediante diferentes pestañas.

5.2.2. Controles

Los controles habituales de la paleta de Visual Studio son:

- ***Label***: es una etiqueta donde podemos mostrar texto por pantalla.
- ***TextBox***: es una caja de texto donde el usuario puede escribir.
- ***PictureBox***: permite mostrar una imagen.
- ***Button***: es un botón que puede pulsar el usuario.
- ***LisBox***: lista de elementos de los cuales el usuario puede seleccionar uno o varios.
- ***ComboBox***: similar al *ListBox*, con la diferencia de que es una lista desplegable, y el usuario sólo puede seleccionar un elemento.
- ***CheckBox***: elemento que el usuario puede marcar como seleccionado, o no. En caso de haber varios *CheckBox* uno bajo el otro, funcionan de forma independiente, de manera que el usuario puede seleccionar tantos como desee.
- ***RadioButton***: elemento similar al *CheckBox*, pero que está diseñado para que el usuario seleccione únicamente uno de entre un conjunto de opciones. A modo de ejemplo, se puede utilizar para preguntar al usuario si está soltero o casado, de manera que tendrá que seleccionar una opción o la otra, pero no ambas al mismo tiempo.
- ***TreeView***: permite mostrar un conjunto de elementos jerárquicos dentro de una estructura de tipo árbol.
- ***DataGridView***: control muy potente, que permite mostrar y gestionar datos en forma de tabla. Es muy útil para hacer el mantenimiento de una tabla de la base de datos.
- ***MenuStrip***: permite crear un menú de forma muy sencilla. Este menú puede contener a su vez otros submenús.
- ***ContextMenuStrip***: permite crear un menú contextual, y asociarlo a un conjunto de elementos. De esta forma, cuando el usuario pulse botón derecho sobre un cierto elemento, se mostrará el menú contextual correspondiente.

5.2.3. Componentes

Asimismo, entre los elementos de la paleta de controles de Visual Studio, hallamos los componentes siguientes:

- ***FileSystemWatcher***: permite vigilar el sistema de ficheros, y reaccionar ante modificaciones que tengan lugar en él. A modo de ejemplo, lo podemos configurar para que nos avise cuando se elimine o modifique un archivo en un cierto directorio.

- ***MessageQueue***: ofrece acceso a un servidor de mensajería de colas, como por ejemplo MSMQ (*Microsoft Message Queuing*).

- ***SerialPort***: nos permite realizar comunicaciones por un puerto serie.

- ***Timer***: lanza un evento cada cierto periodo de tiempo.

5.2.4. Diálogos

También disponemos de un conjunto de diálogos prefabricados, que son muy útiles para tareas cotidianas como elegir la ruta de un archivo, o seleccionar un color de una paleta de controles. Concretamente, disponemos de los siguientes controles:

- ***ColorDialog***: permite seleccionar un color de una paleta de colores.
- ***FolderBrowserDialog***: permite seleccionar un directorio.
- ***FontDialog***: permite seleccionar un tipo de fuente.
- ***OpenFileDialog***: permite abrir un archivo.
- ***SaveFileDialog***: permite grabar en un archivo.
- ***PrintDialog***: permite imprimir por una impresora.
- ***PrintPreviewDialog***: muestra una vista preliminar de impresión.
- ***PageSetupDialog***: permite establecer las medidas de la página, márgenes, etc., antes de imprimir.

En caso de que necesitemos un diálogo que no se corresponda con ninguno de los anteriores, crearemos nuestros propios diálogos a medida. Para ello, basta con añadir nuevos formularios al proyecto, y cuando la aplicación esté en ejecución, iremos mostrando un formulario u otro según nos convenga.

Por último, cabe decir que los diálogos se pueden mostrar de dos formas:

- **Modal**: exige que el usuario responda al diálogo para poder continuar con la ejecución.

- **No modal**: no exige que el usuario responda al diálogo, de manera que se comporta como una ventana independiente. El usuario puede cambiar entre una y otra ventana sin ningún problema.

Para mostrar un diálogo de forma modal, se utiliza el método *ShowDialog*, mientras que para mostrarlo de forma no modal, es con el método *Show*.

6. ASP.NET

En este apartado, se introduce la tecnología ASP.NET, que permite crear aplicaciones web. Tras la visión general sobre ASP.NET y un repaso de algunos de los controles más utilizados, se ofrece una introducción a AJAX, que permite mejorar la experiencia de usuario en las aplicaciones web.

6.1. Una introducción a ASP.NET

ASP.NET es un *framework* para la creación de aplicaciones web, donde se puede programar en cualquiera de los lenguajes de .NET. Apareció en el año 2002, y es la tecnología sucesora de Active Server Pages (ASP) que existe desde 1996.

ASP

Las páginas ASP contienen *scripts* programados habitualmente en VBScript.

ASP.NET ofrece múltiples ventajas en comparación con la antigua ASP:

- ASP.NET se integra totalmente con .NET, y sus páginas se pueden programar en cualquiera de los lenguajes de .NET, haciendo uso de la programación orientada a eventos.
- ASP.NET ofrece un conjunto mucho más rico de controles.
- ASP era interpretado, mientras que ASP.NET es compilado. Esto ofrece múltiples ventajas, como un rendimiento mucho mejor, y una depuración mucho más potente.
- La configuración y despliegue de aplicaciones ASP.NET es mucho más sencillo, ya que la configuración tiene lugar en único archivo texto, y para hacer el despliegue basta con copiar los archivos en el directorio correspondiente.

Las páginas ASP.NET se denominan *web forms* (formularios web), y son archivos con extensión .aspx. Estos archivos están formados básicamente por marcas XHTML estático, y también por marcas ASPX que le dan el comportamiento dinámico. Un formulario web es una clase derivada de *System.Web.UI.Page*, con un conjunto de propiedades, métodos y eventos:

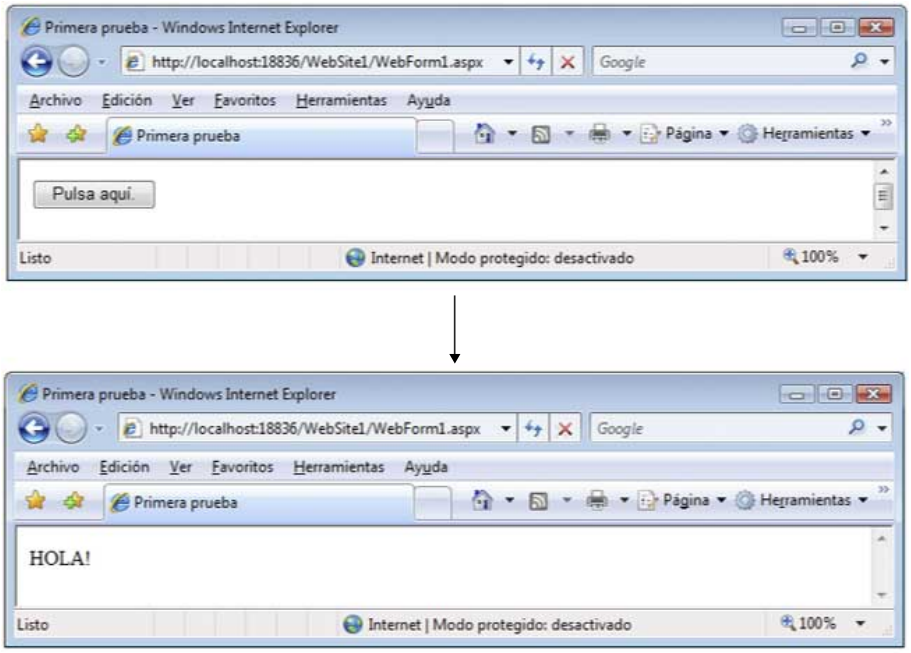
- **Propiedades**: permiten por ejemplo cambiar la apariencia de un formulario (su título, color de fondo, estilos CSS, etc.).
- **Métodos**: exponen el comportamiento del formulario.

- **Eventos**: permiten interactuar con el formulario y asociar el código a ejecutar cuando se produzcan estos eventos.

El diseño y la programación de formularios web son muy similares a WinForms, de manera que un formulario contiene un conjunto de controles que forman la interfaz de usuario, y éstos responden a una serie de eventos a los que se asocia el código correspondiente.

La siguiente página ASP.NET contiene un botón, que al pulsarlo muestra el mensaje "HOLA!":

Figura 9. Primera aplicación ASP.NET



El código correspondiente es el siguiente:

a) WebForm1.aspx: contiene el código XHTML de la página, mezclado con código ASP.NET.

```
<%@ Page Language = "C#" AutoEventWireup="true"
                    CodeFile="WebForm1.aspx.cs" Inherits="WebForm1" %>

<!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>Primera prueba</title>
</head>
<body>
  <form id="form1" runat="server">
    <asp:Button ID="Button1" runat="server" Text="Pulsa aquí."
```

```
      onclick="Button1_Click" />
    <asp:Label ID="Label1" runat="server"></asp:Label>
  </form>
</body>
</html>
```

- El elemento *Page* indica que la página utiliza el lenguaje C#, que deriva de la clase *WebForm1*, y que el código C# se encuentra en el archivo WebForm1.aspx.cs.

- El elemento *DOCTYPE* indica el DTD de XHTML utilizado en la página, que es concretamente XHTML Transitional.

- El elemento *head* hace referencia a la cabecera de la página XHTML, donde se indica, por ejemplo, el título de la página.

- En el cuerpo de la página, hay un formulario que contiene un botón y una etiqueta. En ambos controles podemos ver runat="server", lo que indica que se procesan en el servidor.

- El botón tiene asociada la función *Button1_Click* al evento *onclick*, que se ejecutará cuando el usuario pulse el botón.

b) WebForm1.aspx.cs: contiene el código de los eventos de la página.

```
public partial class WebForm1 : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
  }

  protected void Button1_Click(object sender, EventArgs e)
  {
    Button1.Visible = false;
    Label1.Text = "HOLA!";
  }
}
```

- La clase del formulario deriva de *System.Web.UI.Page*.

- El método *Page_Load* será llamado cada vez que se carga la página.

- El método *Button1_Click* será llamado cuando el usuario pulse el botón *Button1*. En ese momento, ocultaremos el botón y mostraremos "HOLA!" por pantalla.

c) Y por último, el código HTML que ha recibido el navegador:

```
<!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>
  <title>Primera prueba</title>
</head>
<body>
  <form id="form1" method="post" action="WebForm1.aspx">
  <input type="hidden" id="__VIEWSTATE" value="..." />
  <span id="Label1">HOLA!</span>
  </form>
</body>
</html>
```

- El formulario *form1* es de tipo *post* por defecto, y el *action* apunta a la página *WebForm1.aspx*, que será adonde se enviará la solicitud cuando el usuario pulse el botón.

- Toda página contiene, por defecto, un elemento *VIEWSTATE* de tipo *hidden*. Este elemento permite almacenar los datos entrados por el usuario en un formulario, y así se puede persistir el estado de la página entre varias interacciones del usuario.

Evento *Load* del formulario

En el siguiente ejemplo, se muestra un uso típico del evento *Load*:

```
private void Form2_Load(object sender, EventArgs e)
{
   if (!Page.IsPostBack)
      Label1.Text = "Bienvenido";
}
```

- El evento *Load* se ejecuta cada vez que se carga la página. Por tanto, como el *action* del formulario hace referencia a la propia página, esto hace que, cuando el usuario pulsa el botón, el primer evento que se ejecuta es *Page_Load*, y a continuación el *Button1_Click*.

- La propiedad *Page.IsPostBack* es muy útil, y permite diferenciar entre la primera vez que se ejecuta el evento *Page_Load*, y el resto de ocasiones. En el ejemplo anterior, la primera vez que se carga la página se muestra el texto “Bienvenido”.

6.1.1. Controles

ASP.NET ofrece un amplio repertorio de controles que podemos utilizar en los formularios web. A continuación, repasamos algunos de los principales.

Controles estándar

Los controles estándar son los siguientes:

- ***Button***: es un botón que, al ser pulsado, envía la petición correspondiente al servidor.
- ***CheckBox***: permite marcar o desmarcar una cierta opción.
- ***DropDownList***: muy parecido al control ComboBox de WinForms, donde el usuario selecciona un valor de un desplegable.
- ***Image***: permite mostrar una imagen.
- ***Label***: permite mostrar un texto.
- ***ListBox***: permite seleccionar uno o múltiples valores de una lista.
- ***RadioButton***: elemento similar al *CheckBox*, pero con la diferencia de que si hay varios, el usuario sólo puede seleccionar uno de ellos.
- ***TextBox***: permite que el usuario escriba un texto.

Controles de datos

Los controles de datos son un conjunto de controles encargados de la conexión con el origen de datos, y también de la consulta y manipulación de dichos datos:

- ***SqlDataSource***: permite conectar con cualquier base de datos relacional a la que se pueda acceder mediante un proveedor ADO.NET, ya sea ODBC, OLE DB, SQL-Server u Oracle. Su objetivo es reemplazar el código ADO.NET necesario para establecer una conexión y definir la sentencia SQL a ejecutar.
- ***XmlDataSource***: permite enlazar con un documento XML, y realizar consultas mediante *XPath*.
- ***GridView***: es el sucesor del control *DataGrid* de ASP.NET, y permite editar datos en forma tabular. De forma sencilla, podemos modificar o eliminar datos, seleccionar una cierta fila, paginar, etc.
- ***DetailsView***: control similar al *GridView*, pero que sólo permite trabajar con un único registro al mismo tiempo.

Controles de validación

Los controles de validación son un conjunto de controles que facilitan la validación de los datos entrados por el usuario en un formulario. Para mayor seguridad, estos controles realizan una primera validación en el navegador por medio de Javascript, y posteriormente una segunda validación en el servidor. Son los siguientes:

- ***RequiredFieldValidator***: asegura que el usuario no pueda dejar en blanco un campo obligatorio.

- ***RangeValidator***: comprueba que el valor entrado por el usuario esté dentro de un cierto rango de valores.

- ***RegularExpressionValidator***: comprueba que el valor entrado por el usuario cumpla con una cierta expresión regular.

Controles de *login*

Los controles de *login* son un conjunto de controles encargados de las funcionalidades de autenticación de usuarios, como la entrada del usuario y contraseña, creación de usuarios, cambio de contraseña, restauración de la contraseña en caso de pérdida, etc.

Controles de navegación

Los controles de navegación son los controles que facilitan la navegación del usuario por la aplicación:

- ***SiteMapPath***: muestra un conjunto de enlaces representando la página actual del usuario, y el camino jerárquico de vuelta a la página raíz de la web.

- ***Menu***: permite añadir un menú a la aplicación.

- ***TreeView***: permite representar un conjunto de elementos organizados jerárquicamente en forma de árbol.

Controles de *webparts*

Nuevo conjunto de controles disponibles a partir de ASP.NET 2.0, que permiten al usuario personalizar el contenido y aspecto de las páginas web. Algunos de estos controles son los siguientes:

- ***WebPartManager***: gestiona todos los *webparts* de una página.

- ***CatalogZone***: permite crear un catálogo de controles *webpart* que el usuario puede seleccionar y añadir a la página.

- ***EditorZone***: permite al usuario editar y personalizar los controles *webpart* de una página.

- ***WebPartZone***: contiene todos los controles *webpart* que se hayan incluido en la página.

Controles HTML

Los controles HTML son controles que permiten trabajar con elementos simples HTML como tablas, enlaces, imágenes, botones, etc. La diferencia con los controles estándar de ASP.NET es que los HTML generan el mismo código HTML, independientemente del navegador del cliente, mientras que los ASP.NET generan un código HTML en función del navegador del cliente.

Controles de usuario

Por último, también podemos crear nuestros propios controles de usuario de forma muy sencilla. Para ello, hay que crear un proyecto web de tipo ASP.NET, y añadir un nuevo elemento de tipo *Web User Control*. Esto añade al proyecto un archivo .ascx, que será donde crearemos el nuevo control de usuario. Una vez acabado el trabajo, al compilar el proyecto obtendremos un archivo con extensión .dll correspondiente al nuevo control de usuario, que podremos añadir a la paleta de controles de Visual Studio.

6.2. AJAX

Históricamente, las aplicaciones de escritorio siempre han ofrecido una interacción con el usuario mucho más rica que las aplicaciones web. En contrapartida, las aplicaciones web no se tienen que instalar, y nos aseguran que los usuarios siempre tienen la última versión de la aplicación, ya que tan sólo hay que actualizarla una única vez en el servidor.

En cambio, con la aparición de AJAX el tema ha cambiado, ya que es posible crear aplicaciones web que ofrezcan una rica interacción con el usuario. En realidad, las tecnologías que han habilitado la aparición de AJAX existen desde hace más de una década, ya que se trata básicamente de Javascript y XML. No obstante, es reciente el hecho de que se les ha sacado todo el partido con aplicaciones como Google Gmail, Microsoft Outlook Web Access, Flickr, etc.

6.2.1. Una introducción a AJAX

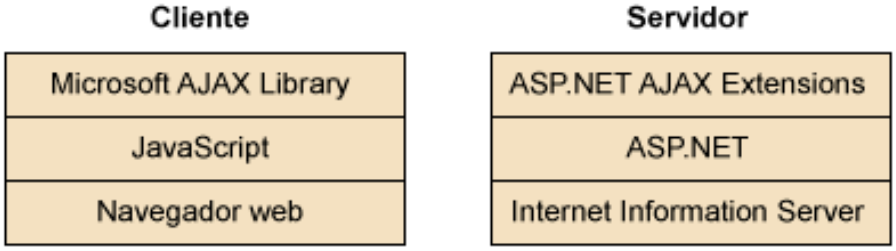
Al desarrollar una aplicación AJAX[9], en lugar de diseñarla como una serie de páginas enlazadas (donde al acabar de procesar una página, simplemente se muestra la siguiente), se utiliza Javascript para mantener una comunicación asíncrona con el servidor web, y así poder actualizar **partes** de las páginas de forma dinámica.

(9)Del inglés, *Asynchronous JavaScript and XML*.

A modo de ejemplo, si el usuario selecciona un país en un desplegable de países, de forma transparente al usuario tiene lugar una comunicación con el servidor donde se le envía el país y el servidor devuelve una lista de ciudades de dicho país. Posteriormente, podemos solicitar al usuario que seleccione una ciudad de dicho país (ello habrá tenido lugar sin haber recargado la página completa, que hubiera sido el procedimiento habitual).

ASP.NET AJAX es el nombre de la solución AJAX de Microsoft, y hace referencia a un conjunto de tecnologías de cliente y de servidor que facilitan el uso de tecnología AJAX en .NET. La siguiente figura muestra, de forma esquemática, los componentes de las aplicaciones AJAX en .NET:

Figura 10. Componentes de AJAX en el cliente y servidor



Por un lado, en el cliente disponemos de Microsoft AJAX Library, una librería Javascript que funciona en múltiples navegadores, y que facilita el desarrollo con Javascript. Lo utilizaremos para interactuar con el DOM, actualizar porciones de la página dinámicamente, realizar comunicaciones asíncronas con el servidor, etc. De hecho, la gran ventaja de Microsoft AJAX Library es que ofrece una capa orientada a objetos de alto nivel, que evita toda la tediosa programación JavaScript a bajo nivel.

Por otro lado, en el servidor disponemos de ASP.NET AJAX Extensions, que está construido por encima de los controles y clases de ASP.NET, y nos permite hacer uso de la Microsoft AJAX Library.

6.2.2. Actualizaciones parciales de páginas

El control *UpdatePanel* es un control contenedor que permite marcar las porciones de una página a actualizar asíncronamente. Cuando alguno de los controles de dentro del *UpdatePanel* genere un envío al servidor (por ejemplo la

pulsación de un botón), podemos iniciar una comunicación asíncrona con el servidor y actualizar únicamente esa porción de página. El hecho de que sea asíncrono quiere decir que el usuario podrá seguir utilizando la página e interactuar con el resto de controles mientras esperamos recibir la respuesta del servidor. En cuanto tengamos la respuesta, actualizaremos la porción de página correspondiente de una forma muy suave a la vista, de manera que el usuario no notará ningún parpadeo ni recarga de la página.

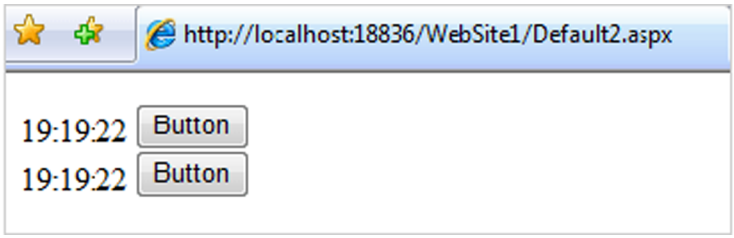
A continuación, se indican los pasos a seguir para realizar un ejercicio muy simple con AJAX, y empezar a entender su funcionamiento más básico:

1) Crear un proyecto web de tipo ASP.NET.
2) Añadir al proyecto una página de tipo *Web Form*.
3) Añadir a la página un control de tipo *ScriptManager*, que es necesario en toda página con AJAX, ya que es el encargado de gestionar el refresco parcial de la página.
4) Añadir debajo un control *Label*, y a su lado un control *Button*.
5) Añadir debajo un control *UpdatePanel*, y dentro de este control, añadir otro nuevo control *Label*, y a su lado otro *Button*.
6) Hacer doble click sobre el formulario, y en el evento *Page_Load* añadir el siguiente código:

```
protected void Page_Load(object sender, EventArgs e)
{
  Label1.Text = DateTime.Now.ToLongTimeString();
  Label2.Text = DateTime.Now.ToLongTimeString();
}
```

Al ejecutar la página, deberíamos obtener algo así como:

Figura 11. Ejemplo sencillo con un *UpdatePanel*



En la página, tenemos dos etiquetas y dos botones; la etiqueta y botón de la parte inferior están ubicados dentro del control *UpdatePanel*. Por tanto, al pulsar el botón superior observaremos que tiene lugar una petición normal al servidor, y se refresca la página entera, mostrando las dos horas actualizadas.

En cambio, al pulsar el botón inferior, se inicia una comunicación asíncrona con el servidor, que devuelve el valor de la hora inferior, y únicamente ésta es actualizada por el navegador. Habremos podido observar cómo, en este último caso, la actualización ha sido muy suave, y no se ha producido ninguna recarga de la página.

Reflexión

Las posibilidades de Microsoft AJAX Library son extensas y quedan fuera del alcance de este libro. Para más información, consultad la bibliografía recomendada.

6.2.3. AJAX Control Toolkit

Sobre AJAX Control Toolkit

Se puede encontrar toda la información sobre el proyecto AJAX Control Toolkit (accesible en línea) en su página web.

Al acceder a esta página, es posible consultar un listado de todos los controles del Toolkit, junto con una demostración de su funcionamiento e información explicativa.

AJAX Control Toolkit es un proyecto de código abierto donde contribuyen empleados de Microsoft y también de otras empresas, cuyo objetivo es crear un rico conjunto de controles que hagan uso de AJAX.

El Toolkit incorpora tanto nuevos controles como extensores que se asocian a controles ya existentes, y añaden diversas características AJAX.

Veamos algunos de los controles del Toolkit:

- ***Animation***: permite realizar múltiples animaciones en función de las acciones del usuario. Es posible mover un elemento, cambiarlo de tamaño, ocultarlo, etc.

- ***Accordion***: permite tener un conjunto de paneles, y mostrar uno u otro con un efecto similar al utilizado en Microsoft Outlook.

- ***AlwaysVisibleControl***: permite hacer que un control sea siempre visible en un posición fija de pantalla, independientemente de que el usuario haga *scroll* arriba o abajo.

- ***Calendar***: aparece una ventana emergente con un calendario, y permite al usuario seleccionar una fecha.

- ***CollapsiblePanel***: control similar al *Accordion*, pero que únicamente trabaja con un panel, que se muestra u oculta pulsando en un botón. Resulta muy útil para ocultar una porción de página que, en ocasiones, el usuario no deseará consultar.

- ***ConfirmButton***: muestra un diálogo emergente que nos solicita confirmar una acción.

- ***ResizableControl***: permite que un elemento pueda ser cambiado de tamaño por el usuario.

- ***Tabs***: permite mostrar el contenido de la página clasificado en diferentes pestañas.

7. WPF

En este apartado, se introduce la nueva tecnología WPF.

WPF (*Windows Presentation Foundation*) está basado en XML, y permite crear potentes interfaces de usuario, tanto en aplicaciones de escritorio como para entornos web (WPF Browser o Silverlight).

WPF permite diseñar interfaces gráficas de usuario más espectaculares que WinForms, pero no está pensado para aplicaciones que requieran un uso intensivo de gráficos o 3D, como por ejemplo juegos. En este campo, las tecnologías más adecuadas siguen siendo DirectX u OpenGL.

WPF apareció en el año 2006, formando parte de .NET Framework 3.0 y Windows Vista. WPF es la evolución de Windows Forms, que probablemente con el tiempo sea descontinuado por Microsoft, para centrarse en las mejoras para WPF.

Aunque WPF está pensado para crear aplicaciones de escritorio, existen variantes para hacer uso de WPF en entornos web:

- **XBAP o WPF Browser**: son aplicaciones WPF que se ejecutan dentro de un navegador web (actualmente, Internet Explorer o Firefox) en entorno Windows. Se pueden utilizar todas las características de WPF, pero hay que tener en cuenta que, al estar dentro de un navegador, las aplicaciones están en un entorno de seguridad restringido.

- **Silverlight**: es un subconjunto de WPF para la creación de aplicaciones web. A diferencia de XBAP, Silverlight está diseñado para ser multiplataforma. Otra diferencia es que Silverlight es muy ligero, con lo que el *plug-in* requerido es mucho más rápido de instalar.

Silverlight para Linux

Ya está disponible Moonlight, que es una implementación de código abierto de Silverlight para sistemas Linux.

7.1. Características

Algunas de las características más interesantes de WPF son las siguientes:

- WPF integra múltiples tecnologías como, por ejemplo, procesamiento gráfico 2D, 3D, vídeo, audio, etc.

- WPF utiliza internamente DirectX, por lo cual se beneficia de todas sus características de gráficos vectoriales, características avanzadas de visualización, técnicas 3D, aceleración por hardware, etc.

- Las interfaces gráficas de WPF se pueden definir, de forma declarativa, con el lenguaje XAML (eXtensible Application Markup Language), lo que permite separar la interfaz gráfica de la lógica de la aplicación.

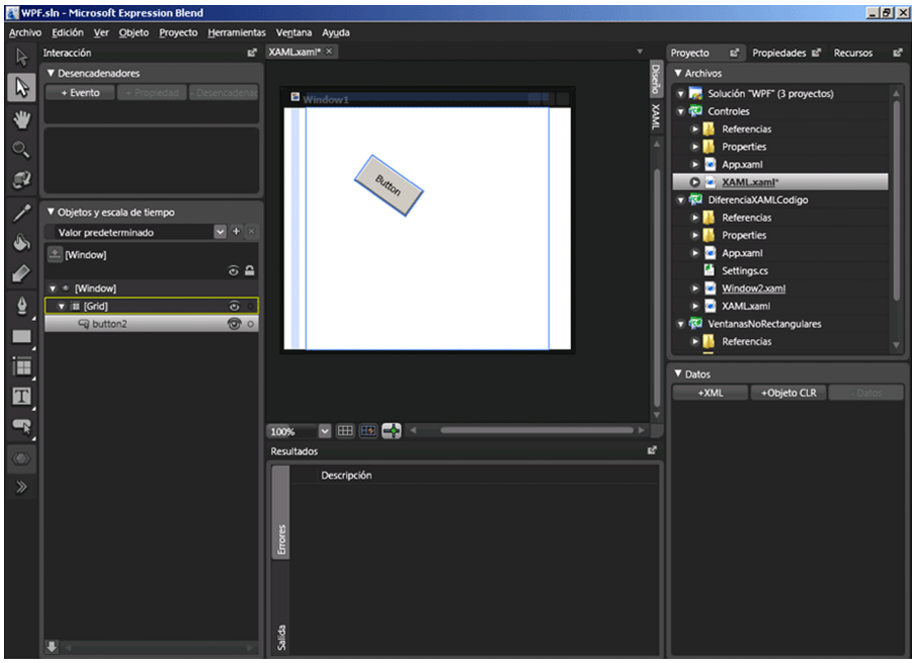
- WPF es interoperable con WinForms y con Win32.

- WPF permite una fácil implantación, ya que al ser .NET basta simplemente con copiar los ensamblados.

7.1.1. Herramientas de desarrollo

Si bien el entorno de programación WPF es Visual Studio, Microsoft ha introducido el entorno de diseño Expression Blend pensado para los diseñadores gráficos; su uso no requiere tener conocimientos de programación.

Figura 12. Expression Blend



De hecho, Expression Blend forma parte del paquete de herramientas de diseño llamado Microsoft Expression Studio que incluye:

- **Expression Web**: herramienta para la creación de interfaces web con HTML. Es la aplicación sucesora de FrontPage, que ya ha sido descontinuado por parte de Microsoft.

- **Expression Blend**: herramienta para la creación de interfaces gráficas de usuario en XAML para aplicaciones WPF o Silverlight.

- **Expression Design**: herramienta para la creación de gráficos vectoriales. Ofrece una buena integración con Expression Blend gracias a la importación/exportación de código XAML.

- **Expression Media**: es un organizador de contenido multimedia (fotos, vídeos o audio). Permite categorizar y añadir metadatos a los diferentes archivos.

- **Expression Encoder**: aplicación para la recodificación de vídeos. Permite elegir un nivel de calidad predefinido, un códec, y formato de salida, y previsualizar el resultado.

7.1.2. XAML

XAML es un lenguaje XML que permite describir una interfaz gráfica WPF. Pero no todas las aplicaciones WPF necesariamente han de utilizar XAML, ya que es posible definir interfaces gráficas de forma procedural (con código fuente en algún lenguaje de .NET). Veamos un ejemplo con un botón:

```
Button b = new Button();
b.Name = "button1";
b.Content = "Pulsa aquí";
b.Height = 23;
```

El código anterior es equivalente al siguiente fragmento de XAML:

```
<Button Height="23" Name="button1">
  Pulsa aquí
</Button>
```

Uno de los motivos para utilizar XAML es que permite separar la interfaz gráfica, de la lógica de aplicación. Esta separación permite modificar la visualización de la interfaz sin afectar a la lógica de la aplicación y viceversa.

De todas formas, aunque XAML es un lenguaje declarativo, a la hora de compilar una aplicación WPF, el código XAML se convierte a una representación binaria llamada BAML (*Binary Application Markup Language*), más eficiente y compacta, que se incluye como recurso dentro del ensamblado final.

Hay diversas variantes de XAML; algunas de las más conocidas son:

- XAML para interfaces gráficas WPF.

- XAML para definir el formato de documentos electrónicos XPS.
- XAML para Silverlight.
- XAML para WWF, utilizado para definir los flujos de trabajo.

Formato XPS

Del inglés *XML Paper Specification*. Es un formato de documentos desarrollado por Microsoft, y similar a PDF. Windows Vista dispone de una impresora XPS, y también de un visor de XPS integrado en Internet Explorer.

7.1.3. Window

El elemento principal de una aplicación WPF es la ventana, representada por la clase *Window*. Al crear un proyecto WPF en Visual Studio, se crea automáticamente un fichero xaml con un elemento *Window*:

```
<Window x:Class="Layout.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Window1" Height="300" Width="300">
  <!-- Contenido de la ventana -->
</Window>
```

En WPF, los controles no están sujetos a una resolución de pantalla concreta, de forma que podemos escalar una interfaz sin perder calidad de visualización, y manteniendo la distribución de los controles. Para organizar los controles, se utilizan unos contenedores llamados paneles, que distribuyen los controles de su interior siguiendo un determinado patrón.

7.1.4. Controles contenedores

A continuación, veremos algunos de los principales controles contenedores de WPF, como son el *Grid*, *Canvas* y *StackPanel*.

Grid

El *Grid* es un contenedor muy flexible en cuanto a la organización de los controles, ya que permite definir varias filas y columnas en forma de tabla, donde posteriormente se ubican los diferentes controles.

Al principio, hay una sección *RowDefinitions* y *ColumnDefinitions*, donde se definen el número de filas y columnas, asi como sus medidas o proporciones. A continuación, ubicamos cada control en la fila o columna que corresponda, con las propiedades *Grid.Row* y *Grid.Column*. Además, con las propiedades *RowSpan* y *ColumnSpan* podemos indicar si un control ocupa varias filas o columnas:

Proporciones

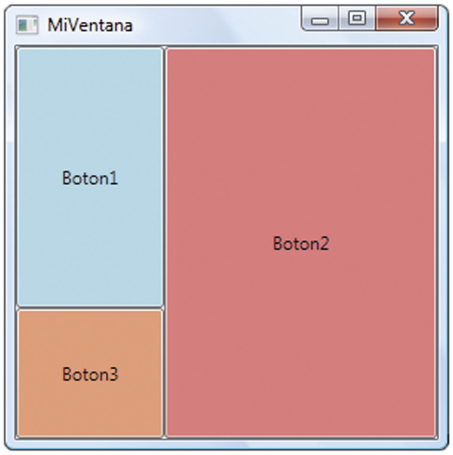
Las proporciones se indican con el carácter asterisco (*). En el ejemplo, la primera fila tiene una altura de 2*, lo que significa que será el doble de alta que el resto.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="2*"/>
```

```
    <RowDefinition Height="*"/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Button Name="Button1" Content="Boton1" Grid.Row="0"
      Grid.Column="0" Background="LightBlue"/>
  <Button Name="Button2" Content="Boton2" Grid.Row="0"
      Grid.Column="1" Background="LightCoral"
      Grid.RowSpan="2"/>
  <Button Name="Button3" Content="Boton3" Grid.Row="1"
      Grid.Column="0" Background="LightSalmon"/>
</Grid>
```

Al ejecutar la aplicación, podemos comprobar que el botón 2 ocupa las dos filas debido al *RowSpan*, y el botón 1 es el doble de alto que el botón 3:

Figura 13. Ejemplo de panel Grid



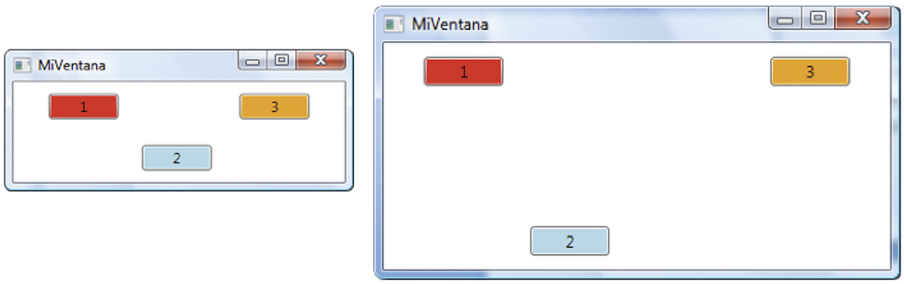
Canvas

En un panel de tipo *Canvas* los controles se sitúan en relación a su distancia con alguna de las esquinas del panel. Veamos un ejemplo:

```
<Canvas Name="canvas1">
  <Button Canvas.Left="30" Canvas.Top="10"
      Background="Red" Width="60">1</Button>
  <Button Canvas.Left="110" Canvas.Bottom="10"
      Background="LightBlue" Width="60">2</Button>
  <Button Canvas.Right="30" Canvas.Top="10"
      Background="Orange" Width="60">3</Button>
</Canvas>
```

Al ejecutar la aplicación, podemos comprobar cómo se mantienen las distancias a los bordes independientemente del tamaño de la ventana:

Figura 14. Ejemplo de panel Canvas



StackPanel

El panel *StackPanel* distribuye los controles ubicados en su interior en forma de pila, secuencialmente. El siguiente ejemplo utiliza un *StackPanel* para distribuir diferentes botones:

```
<StackPanel Name="stack1" Orientation="Vertical">
   <Button Background="Red">      1 </Button>
   <Button Background="Orange">   2 </Button>
   <Button Background="Yellow">   3 </Button>
   <Button Background="Blue">     4 </Button>
</StackPanel>
```

La propiedad *Orientation* del panel indica la orientación que toman los componentes, que puede ser Vertical o Horizontal. La siguiente figura muestra el resultado del código anterior con los dos tipos de orientación:

Figura 15. Ejemplo de panel *StackPanel*

ScrollViewer

El control *ScrollViewer* no es un panel en sí, pero se puede combinar con un panel para añadir la funcionalidad de barras de desplazamiento a los componentes que contiene.

El control *ScrollViewer* puede utilizarse no sólo con paneles sino también con otros controles; por ejemplo, el siguiente código utiliza un *ScrollViewer* para hacer *scroll* de una imagen:

```
<ScrollViewer HorizontalScrollBarVisibility="Auto">
  <Image Source="flor_almendro.jpg"/>
</ScrollViewer>
```

7.1.5. Eventos

En WPF, todos los controles disponen de un evento *Loaded*, que corresponde a cuando el control ha sido inicializado y está listo para ser renderizado. Un evento muy utilizado es el *Window_Loaded*, ya que es el evento que aparece por defecto al pulsar sobre el fondo de una ventana en Visual Studio:

```
private void Window_Loaded(object o, RoutedEventArgs e)
{
}
```

Aparte de este evento, de forma muy similar a Windows Forms, los controles WPF disponen de un amplio abanico de eventos como, por ejemplo, al pulsar el botón del *mouse*, pulsar una tecla, ganar/perder el foco, etc.

WPF incorpora como novedad el enrutamiento de eventos, que ofrece gran flexibilidad para gestionar los eventos del árbol visual de elementos que forman la interfaz de usuario.

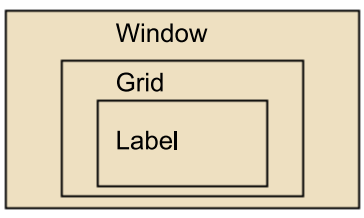
Todos los controles de una ventana o página WPF forman una estructura jerárquica denominada árbol visual.

Árbol visual de elementos

A modo de ejemplo, mostramos el código XAML de una página, con el árbol visual correspondiente:

```
<Window>
  <Grid>
    <Label>
    </Label>
  </Grid>
</Window>
```

Figura 16. Árbol visual de elementos



En WPF hay tres tipos de eventos:

- ***Direct***: es el funcionamiento clásico que había en Windows Forms, donde sólo se notifica al propio elemento que originó el evento.

- ***Tunnel***: el evento traviesa el árbol visual de arriba abajo, empezando por la raíz, y acabando en el elemento que originó el evento. Por convención, estos eventos tienen el prefijo *Preview*.

- ***Bubble***: el evento atraviesa el árbol visual de abajo arriba, empezando en el elemento que originó el evento, y acabando en la raíz.

En el siguiente ejemplo, mostramos los eventos en funcionamiento. Para ello, asociamos una misma función genérica a los eventos *MouseDown* (de tipo *Bubble*) y *PreviewMouseDown* (de tipo *Tunnel*) de los elementos *Window*, *Grid* y *Label*, y observaremos en qué orden se ejecutan las funciones:

- Código del archivo Eventos.xaml:

```
<Window Name="window1"
  PreviewMouseDown="GenericRoutedEventHandler"
  MouseDown="GenericRoutedEventHandler">

  <Grid Name="grid1"
    PreviewMouseDown="GenericRoutedEventHandler"
    MouseDown="GenericRoutedEventHandler">

    <Label Content="Pulsar aquí" Name="label1"
      PreviewMouseDown="GenericRoutedEventHandler"
      MouseDown="GenericRoutedEventHandler">
    </Label>
  </Grid>
</Window>
```

- Código del archivo Eventos.xaml.cs:

```
private void GenericRoutedEventHandler(object sender, RoutedEventArgs e)
{
  string name = ((FrameworkElement)sender).Name;
  Console.WriteLine(name + "\t" +
    e.RoutedEvent.Name + "\t" +
    e.RoutedEvent.RoutingStrategy);
}
```

Al ejecutar la aplicación y pulsar con el botón izquierdo del ratón sobre el elemento *label1*, obtenemos la siguiente salida:

```
window1   PreviewMouseDown    Tunnel
grid1     PreviewMouseDown    Tunnel
label1    PreviewMouseDown    Tunnel
label1    MouseDown           Bubble
grid1     MouseDown           Bubble
window1   MouseDown           Bubble
```

En primer lugar, se ha producido el evento *PreviewMouseDown* que, al ser de tipo *Tunnel*, ha empezado su recorrido en la raíz del árbol (la ventana), luego *grid1* y por último *label1*. En cambio, el evento *MouseDown* sigue el orden inverso, ya que empieza en el elemento donde se ha originado el evento (*label1*), y acaba en la ventana.

Merece la pena comentar que el ejemplo anterior ha sido diseñado para mostrar la propagación de eventos en WPF, pero en las aplicaciones WPF lo más usual es parar la propagación del evento en uno u otro punto.

7.2. Controles

El tipo y la función de los controles predefinidos de WPF es muy similar al de de WinForms, aunque los de WPF ofrecen mucha más flexibilidad. Por ejemplo, los controles de WPF permiten añadir, como contenido de un control, otros controles. En realidad, cada control sólo puede tener un control en su interior, pero si este control es un panel, dentro del mismo se pueden organizar nuevos controles, con lo que se multiplican las combinaciones posibles. A continuación, repasaremos los controles principales de WPF:

Más controles

Aparte de los controles incluidos en el .NET Framework, podemos añadir muchos otros controles (los hay tanto gratuitos como comerciales), o crear nuestros propios controles.

- ***Label***: muestra un texto al usuario.

- ***TextBox***: es el control más común de introducción de datos.

- ***PasswordBox***: permite al usuario introducir *passwords* de forma segura. El *password* introducido por el usuario se puede obtener a través de la propiedad *Password*, que nos devuelve un objeto *SecureString* (es como un *String*, pero se almacena cifrado en memoria).

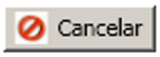
- ***Button***: es un botón. Su contenido no tiene por qué ser exclusivamente texto, sino que podemos poner cualquier tipo de contenido.

Ejemplo

En el siguiente ejemplo, se utiliza un *StackPanel* para agrupar una imagen y una etiqueta dentro del botón:

```
<Button Name="button1" Height="23" Width="73">
  <StackPanel Orientation="Horizontal">
    <Image Source="cancelar.jpg" Width="24"/>
    <Label Content="Cancelar"/> </StackPanel>
  </Button>
```

Figura 17. Botón con imagen y texto



- ***CheckBox***: es un botón que puede estar marcado, o no, y obtenemos este valor con la propiedad *IsChecked*.

- ***RadioButton***: es similar al *CheckButton*, pero con la diferencia de que mientras varios *CheckBox* pueden estar activados a la vez, sólo un *RadioButton* de entre los pertenecientes a un mismo grupo puede estar activado. En caso de no haber ningún grupo, la agrupación se haría entre todos los *RadioButton* de un mismo panel.

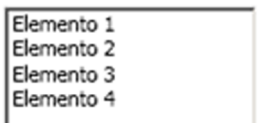
- ***RichTextBox***: es similar al *TextBox*, pero más potente, ya que permite editar texto con formato.

- ***ListBox***: muestra un conjunto de elementos en forma de lista.

Ejemplo de *ListBox*

```
<ListBox Name="listBox1">
  <ListBoxItem>Elemento 1</ListBoxItem>
  <ListBoxItem>Elemento 2</ListBoxItem>
  <ListBoxItem>Elemento 3</ListBoxItem>
  <ListBoxItem>Elemento 4</ListBoxItem>
</ListBox>
```

Figura 18. Control *ListBox*

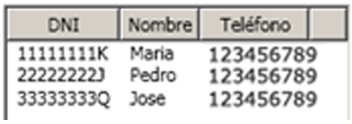


- ***ComboBox***: similar al *ListBox*, con la diferencia de que es una lista desplegable, y el usuario sólo puede seleccionar un elemento.

- ***ListView***: permite mostrar datos de una lista en diversas columnas.

Ejemplo de *ListView*

Figura 19. Control *ListView*

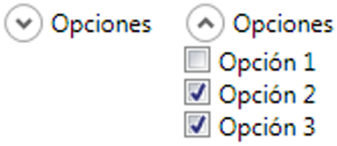


- ***Expander***: permite agrupar un conjunto de controles, y el usuario lo puede contraer o expandir pulsando un botón.

Ejemplo de *Expander*

```
<Expander Header="Opciones">
  <StackPanel>
    <CheckBox>Opción 1</CheckBox>
    <CheckBox>Opción 2</CheckBox>
    <CheckBox>Opción 3</CheckBox>
  </StackPanel>
</Expander>
```

Figura 20. Expander contraído y expandido



- ***TabControl***: permite organizar contenido en forma de pestañas. Los elementos de un control *TabControl* son de tipo *TabItem*; cada uno de ellos corresponde con una pestaña del control.

Ejemplo de *TabControl*

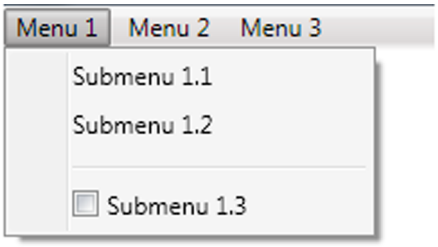
```
<TabControl>
  <TabItem Header="Tab 1">Este es el tab 1</TabItem>
  <TabItem Header="Tab 2">Este es el tab 2</TabItem>
  <TabItem Header="Tab 3">Este es el tab 3</TabItem>
</TabControl>
```

- ***Menu***: permite crear menús fácilmente. Dentro del *Menu* se añade un elemento *MenuItem* para cada menú que se quiera crear, y podemos añadir una barra separadora con un elemento *Separator*.

Ejemplo de *Menu*

```
<Menu Height="22" Name="menu1" VerticalAlignment="Top">
  <MenuItem Header="Menu 1">
    <MenuItem Header="Submenu 1.1"/>
    <MenuItem Header="Submenu 1.2"/>
    <Separator/>
    <CheckBox Content="Submenu 1.3"/>
  </MenuItem>
  <MenuItem Header="Menu 2"/>
  <MenuItem Header="Menu 3"/>
</Menu>
```

Figura 21. Control *Menu*

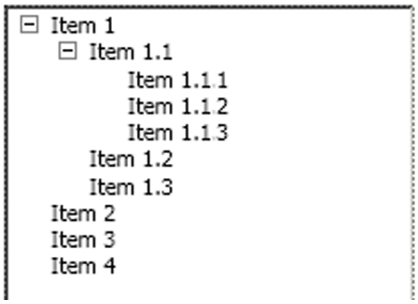


- ***ContextMenu***: tipo especial de menú que se abre al pulsar el botón secundario del ratón encima de algún control.

- ***TreeView***: permite mostrar un conjunto de elementos de forma jerárquica, es decir, en forma de árbol.

Ejemplo de *TreeView*

Figura 22. Control *TreeView*

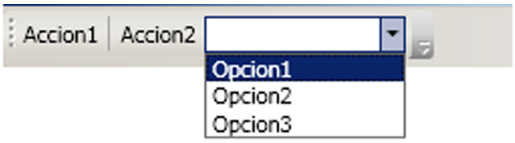


- ***ToolBar***: muestra una barra de iconos o herramientas, formada por botones, etiquetas u otros controles que queramos añadir. El control *ToolBar* se puede añadir en cualquier parte de la interfaz gráfica, aunque generalmente se incluye en un *ToolBarTray*, el cual puede contener diversas barras de herramientas.

Ejemplo de *ToolBar*

```
<ToolBarTray VerticalAlignment="Top">
  <ToolBar Name="toolBar1">
    <Button>Accion1</Button>
    <Separator/>
    <Button>Accion2</Button>
    <ComboBox Width="100">
      <ComboBoxItem>Opcion1</ComboBoxItem>
      <ComboBoxItem>Opcion2</ComboBoxItem>
      <ComboBoxItem>Opcion3</ComboBoxItem>
    </ComboBox>
  </ToolBar>
</ToolBarTray>
```

Figura 23. Control *ToolBar*



- ***ProgressBar***: muestra una barra de progreso, útil por ejemplo para mostrar el porcentaje finalizado de una determinada tarea. Este control tiene las propiedades *Value*, *Minimum* y *Maximum*, que permiten establecer o consultar el valor actual mostrado en la barra, el valor correspondiente al mínimo (barra vacía), y el valor correspondiente al máximo (barra completa), respectivamente:

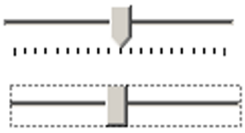
Ejemplo de *ProgressBar*

Figura 24. Control *ProgressBar*

- ***Slider***: permite que el usuario seleccione un valor determinado entre el mínimo y máximo establecidos. La propiedad *TickPlacement* permite visualizar o esconder las marcas de medición.

Ejemplo de *Slider*

Figura 25. Control *Slider* con y sin marcas de medición



- ***WindowsFormsHost***: permite ejecutar cualquier control *WinForms* en una aplicación WPF. Para hacerlo, bastará con añadir a la ventana WPF un control *WindowsFormsHost*, e incluir en su interior el control *WinForms* a ejecutar.

7.3. Funcionalidades gráficas y multimedia

En los subapartados siguientes se representan algunos ejemplos de funcionalidades gráficas y multimedia de WPF.

7.3.1. Transformaciones

WPF permite realizar transformaciones geométricas como rotaciones o traslaciones, a los diferentes elementos de la interfaz. Algunas de las transformaciones más comunes son las siguientes:

- ***RotateTransform***: permite rotar un elemento de la interfaz en un ángulo determinado.

Ejemplo de *RotateTransform*

El siguiente ejemplo muestra cómo rotar un botón 25 grados a la derecha:

```
<StackPanel Width="134">
```

```
    <Button Background="Red" Height="50" Width="100"> 1
    </Button>
    <Button Background="Orange" Height="50" Width="100"> 2
      <Button.RenderTransform>
        <RotateTransform Angle="25" />
      </Button.RenderTransform>
    </Button>
    <Button Background="Yellow" Width="100" Height="50"> 3
    </Button>
  </StackPanel>
```

Figura 26. Rotación de un botón

- ***TranslateTransform***: permite trasladar un elemento siguiendo un cierto vector de traslación, indicado mediante las propiedades X e Y de la clase *TranslateTransform*.

Ejemplo de *TranslateTransform*

Con el siguiente código, trasladamos el botón central del ejemplo anterior:

```
<Button Background="Orange" Height="50" Width="100">
  <Button.RenderTransform>
    <TranslateTransform X="50" Y="-20" />
  </Button.RenderTransform>
  2
</Button>
```

Figura 27. Traslación de un botón

- ***ScaleTransform***: permite escalar un elemento en una cierta proporción determinada por las propiedades siguientes:
  - ScaleX: factor de escalado en horizontal
  - ScaleY: factor de escalado en vertical
  - CenterX: coordenada X del centro de escalado

– CenterY: coordenada Y del centro de escalado

Ejemplo

El siguiente ejemplo realiza un escalado del botón central con factores de escala de 1.5 x 2, y centro de escalado (50,0):

```
<Button Background="Orange" Height="50" Width="100">
  <Button.RenderTransform>
  <ScaleTransform ScaleX="1.5" ScaleY="2" CenterX="50"/>
  </Button.RenderTransform>
  2
</Button>
```

Como se puede comprobar en el resultado, el estiramiento horizontal se realiza de forma uniforme hacia los dos lados, porque la coordenada *x* del centro de escalado se sitúa en el medio del botón, mientras que el escalado vertical se realiza sólo hacia abajo porque la coordenada *y* del centro de escalado se establece en cero por defecto.

Figura 28. Aplicación de *ScaleTransform*

- ***TransformGroup***: permite combinar múltiples transformaciones sobre un elemento, que tienen lugar según el orden en el que se han añadido al *TransformGroup*.

Ejemplo de *TransformGroup*

El siguiente ejemplo aplica una rotación y, a continuación, una translación:

```
<Button Background="Orange" Height="50" Width="100">
  <Button.RenderTransform>
    <TransformGroup>
      <RotateTransform Angle="45"/>
      <TranslateTransform X="50" Y="-50"/>
    </TransformGroup>
  </Button.RenderTransform>
  2
</Button>
```

Figura 29. Resultado de la combinación de rotación y traslación

7.3.2. Animaciones

WPF incorpora diversas clases que permiten crear animaciones, entendiendo por *animación* la modificación de una propiedad durante un determinado intervalo de tiempo y entre un determinado rango de valores.

Las clases se denominan *XXXAnimation*, donde *XXX* es el tipo de dato de la propiedad que se va a animar. WPF soporta los tipos de datos *Char*, *Color*, *Int16*, *Int32*, *Int64*, *Double*, etc.

Ejemplo

El siguiente código hace que el botón cambie su anchura desde 50 a 200 en 2 segundos:

- Código del archivo Window1.xaml

```
<Button Name="boton1" Width="50" Height="50">
 Prueba
</Button>
```

- Código del archivo Window1.xaml.cs

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  DoubleAnimation da = new DoubleAnimation();
  da.From = 50;
  da.To = 200;
  da.Duration = new Duration(new TimeSpan(0, 0, 2));
  boton1.BeginAnimation(Button.WidthProperty, da);
}
```

Figura 30. Inicio y final de la animación

Rotación de una imagen

En el siguiente ejemplo hacemos la rotación de una imagen:

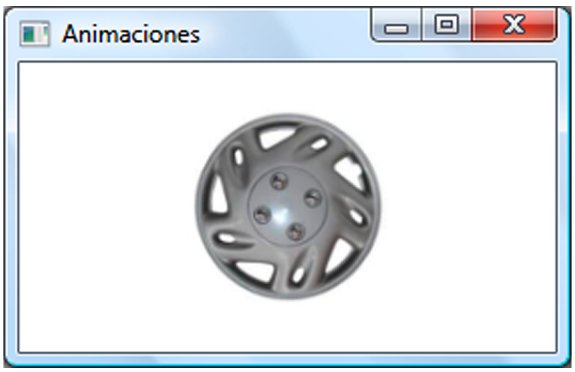
- Código del archivo Window1.xaml

```
<Image Name="imagen" Source="imagen.jpg" Width="100">
</Image>
```

- Código del archivo Window1.xaml.cs

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  DoubleAnimation da = new DoubleAnimation();
  da.From = 0;
  da.To = 360;
  da.Duration = new Duration(new TimeSpan(0, 0, 2));
  da.RepeatBehavior = RepeatBehavior.Forever;
  RotateTransform rt = new RotateTransform();
  imagen.RenderTransform = rt;
  imagen.RenderTransformOrigin = new Point(0.5, 0.5);
  rt.BeginAnimation(RotateTransform.AngleProperty, da);
}
```

Figura 31. Imagen en movimiento



7.3.3. Audio y vídeo

En WPF, reproducir un archivo de audio es tan fácil como incluir el siguiente fragmento de código en la aplicación:

```
System.Media.SoundPlayer sp;
sp = new System.Media.SoundPlayer("sonido.wav");
sp.Play();
```

Y para reproducir vídeo, simplemente tenemos que añadir un control de tipo *MediaElement*, al cual le indicamos en la propiedad *Source* el vídeo a reproducir. Una vez hecho esto, con los métodos *Play*, *Pause* y *Stop* del objeto podremos controlar la reproducción:

```
<MediaElement Name="video1" Source="video.wmv" />
```

7.4. WPF Browser y Silverlight

En este subapartado, trataremos las dos tecnologías que permiten crear aplicaciones web mediante WPF, que son WPF Browser y Silverlight.

7.4.1. WPF Browser

Las aplicaciones WPF Browser se ejecutan a través de un navegador, y tienen acceso a todas las características gráficas y multimedia de WPF, por lo que requieren que el cliente tenga instalado el .NET Framework 3.0. De hecho, en una aplicación WPF Browser podemos hacer casi lo mismo que en una aplicación WPF de escritorio. Una de las mayores diferencias es que las aplicaciones WPF Browser se ejecutan dentro de unos estrictos límites de seguridad, por lo que no tienen acceso ni al registro, ni al sistema de ficheros.

Las URL de las aplicaciones WPF Browser corresponden a un archivo con extensión .xbap y, cuando son accedidas por un navegador, la aplicación entera es descargada y guardada temporalmente en la *cache* del navegador. Una vez descargada en local, la aplicación es automáticamente ejecutada por el navegador, ya que no requiere de ningún proceso de instalación.

Para desarrollar una aplicación WPF Browser con Visual Studio 2008, hay que crear un proyecto de tipo *WPF Browser Application*. Una vez creado, ya podemos programar la aplicación igual como haríamos para WPF, sólo con la diferencia de que, al ejecutarla, se lanza automáticamente un navegador apuntando al archivo .xbap correspondiente a la aplicación compilada.

7.4.2. Silverlight

Silverlight también permite crear aplicaciones web mediante WPF, pero, a diferencia de WPF Browser, Silverlight es únicamente un subconjunto de WPF, por lo que sus funcionalidades son mucho más limitadas.

Más sobre Silverlight

Para más información, podéis acceder a la página web de Silverlight.

Para desarrollar una aplicación Silverlight con Visual Studio 2008, es necesario instalar previamente el paquete denominado "Silverlight Tools for Visual Studio 2008". Una vez instalado, al crear un proyecto de tipo *Silverlight Application*, automáticamente se crea una solución con dos proyectos: un proyecto Silverlight, donde haremos toda la implementación, y un proyecto ASP.NET, que tiene una página HTML con la referencia al archivo .xap de la aplicación Silverlight.

Dos proyectos

El motivo de que se cree un proyecto Silverlight y un proyecto ASP.NET es que, a diferencia de las aplicaciones WPF Browser, las aplicaciones Silverlight deben ser albergadas en una página HTML inicial (similar a como ocurre, por ejemplo, con las *applets* de Java).

8. Windows Mobile

Este apartado ofrece una introducción al desarrollo de aplicaciones para dispositivos móviles, y concretamente para Windows Mobile.

8.1. Una introducción a Windows Mobile

En este subapartado veremos qué dispositivos permiten ejecutar aplicaciones .NET, así como sus sistemas operativos y herramientas de desarrollo disponibles.

8.1.1. Dispositivos

Pocket PC (abreviado, PPC) es una especificación hardware para ordenadores de bolsillo que incorporan el sistema operativo Windows Mobile de Microsoft. Además, estos dispositivos pueden trabajar con múltiples aparatos añadidos, por ejemplo, receptores de GPS, lectores de códigos de barras, cámaras, etc.

Los dispositivos con teléfono y pantalla no táctil se llaman smartphones. La diferencia principal es que un smartphone está pensado para ser un teléfono móvil y utilizarse con una mano, mientras que un *pocket PC* es para hacer de agenda electrónica y usarse con ambas manos. De todas formas, tienden a unificarse, con lo que cada vez es menos clara la diferencia entre ambos.

Figura 32. Ejemplos de *pocket PC* y *smartphones*

Para sincronizar datos y comunicarse, estos dispositivos incorporan mecanismos como USB, WiFi, Bluetooth o IrDa (infrarrojos). La sincronización de los datos con un PC se lleva a cabo con el programa ActiveSync de Microsoft o mediante el centro de sincronización de dispositivos en Windows Vista.

Actualmente, existen miles de aplicaciones comerciales para *pocket PC*, y también podemos desarrollar nuestras propias aplicaciones con C++ Embedded (para obtener el máximo rendimiento), o de forma más fácil con .NET.

8.1.2. Sistemas operativos

Dispositivos con Windows CE

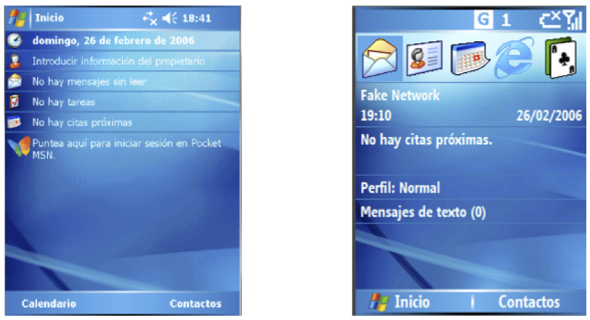
Un ejemplo de dispositivos con Windows CE son los navegadores Tom Tom ONE.

Windows CE es un sistema operativo modular diseñado para dispositivos incrustados, y con una capacidad de proceso y almacenamiento limitada. Al ser modular, Windows CE se ajusta a las necesidades de cada dispositivo, que selecciona y configura únicamente aquellos componentes deseados. Además, los fabricantes acostumbran a incluir sus propias aplicaciones dentro de este sistema.

Hay fabricantes de dispositivos, como por ejemplo Symbol o Intermec, que al comprar los dispositivos nos permiten elegir entre Windows CE y Windows Mobile. Como usuarios finales, podemos detectar fácilmente la diferencia, ya que en Windows CE el botón de inicio acostumbra a estar en la esquina inferior izquierda (como en Windows), mientras que en Windows Mobile el inicio se encuentra en la esquina superior izquierda. Por otro lado, en Windows CE hay barra de tareas, y en Windows Mobile no la hay.

El sistema operativo utilizado por Pocket PCs y Smartphones es Windows Mobile, que es un sistema operativo basado en Windows CE.

Figura 33. Pantalla de inicio de Windows Mobile para *pocket PC* y *smartphones*



En la siguiente lista, se enumeran las versiones más recientes de Windows Mobile, así como la versión de Windows CE en que están basadas:

- Windows Mobile 2003 (basado en Windows CE 4.2)
- Windows Mobile 2003 Second Edition (basado en Windows CE 4.2)
- Windows Mobile 5 (basado en Windows CE 5.0)
- Windows Mobile 6 (basado en Windows CE 5.2)

Concretamente, el último Windows Mobile tiene diferentes ediciones:

- **Windows Mobile Classic** (Pocket PC): para dispositivos sin teléfono, y con pantalla táctil.
- **Windows Mobile Standard** (Smartphone): para dispositivos con teléfono y sin pantalla táctil.

- **Windows Mobile Professional** (Pocket PC Phone Edition): para dispositivos con teléfono y también pantalla táctil.

8.1.3. Herramientas de desarrollo

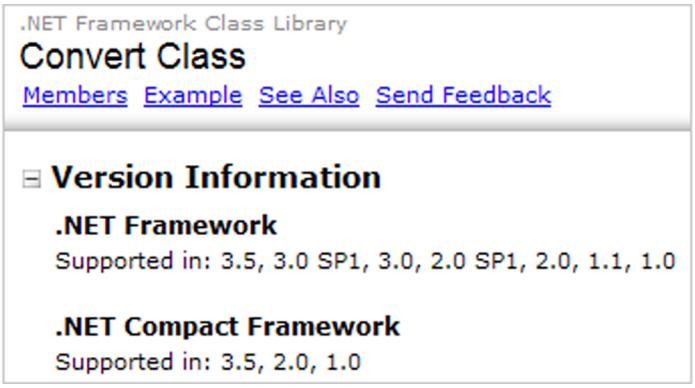
El desarrollo de aplicaciones .NET para Windows Mobile se hace con una versión más limitada del *framework*, llamada .NET Compact Framework (.NET CF), que tiene un número inferior de clases y métodos. El .NET Framework ocupa unos 40 MB de memoria, lo que es demasiado para los *pocket PC*, que habitualmente tienen 64 o 128 MB de RAM. El .NET CF permite reducir el espacio requerido a unos 4 MB de RAM, implementando aproximadamente un 30% de las clases del .NET framework.

En la documentación del .NET Framework se identifican las clases y miembros disponibles para el .NET CF, así como las versiones concretas en que están disponibles.

Documentación de la clase *Convert*

En el siguiente ejemplo, se muestra la documentación de la clase *Convert*:

Figura 34. Documentación del .NET Framework indicando la compatibilidad con .NET CF



Visual Studio facilita el desarrollo de aplicaciones de .NET CF, ya que proporciona diferentes tipos de proyecto y diseñadores específicos para esta plataforma. También permite la depuración y ejecución de aplicaciones de .NET CF de forma integrada, con los emuladores de cada plataforma.

Figura 35. Emuladores de Windows Mobile para *pocket PC* y *smartphone*

8.2. WinForms para dispositivos móviles

El desarrollo de aplicaciones WinForms para dispositivos móviles está optimizado para mejorar el rendimiento y reducir el tamaño. Por este motivo, se han eliminado múltiples funcionalidades poco usables para dispositivos móviles, como *drag&drop*, impresión, controles ActiveX, etc.

8.2.1. Primera aplicación con .NET CF

Para crear una aplicación para dispositivos móviles usando .NET CF, crearemos un proyecto de tipo *Smart Device* en Visual Studio. A continuación, seleccionaremos la plataforma de destino (*pocket PC*, *smartphone*, etc.), la versión de .NET CF, y el tipo de proyecto a crear.

¡Atención!

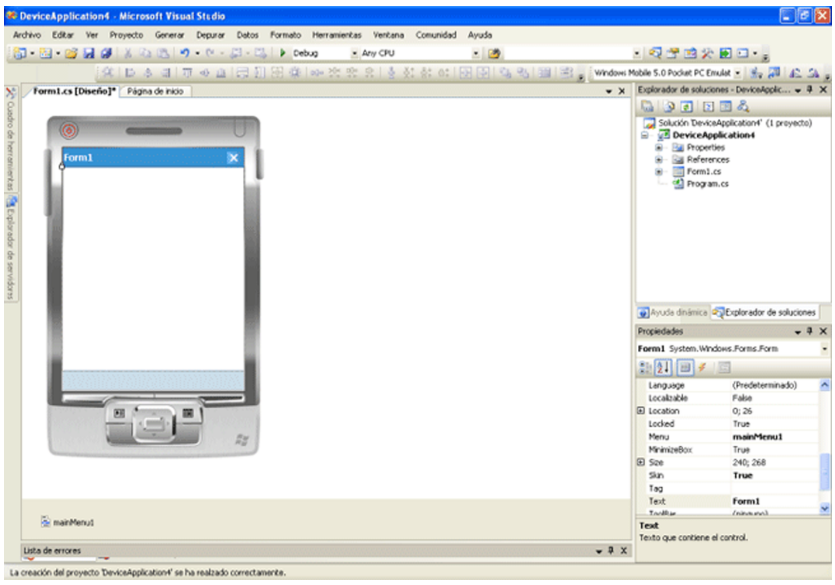
En caso de no aparecer ninguna plataforma de Windows Mobile 6, tendremos que descargar e instalar el Windows Mobile 6 SDK.

Visual Studio añade automáticamente diversos ficheros al proyecto, concretamente Program.cs (programa principal), y Form1.cs (un formulario). Inmediatamente después, se abre el diseñador de formularios específico para la plataforma que hayamos seleccionado, donde podremos arrastrar los controles e ir añadiendo el código correspondiente a los diferentes eventos.

Proyecto para *pocket PC*

En el siguiente ejemplo, hemos creado un proyecto para *pocket PC*.

Figura 36. Diseñador de WinForms para dispositivos móviles



8.2.2. Formularios WinForms

La interacción con los formularios es diferente en función de si se trata de un dispositivo *pocket PC* o un *smartphone*:

1) Pocket PC

Los dispositivos *pocket PC* permiten interactuar con las aplicaciones mediante un lápiz digital utilizado a modo de ratón, por lo que la entrada de datos se produce mediante un teclado mostrado en pantalla o utilizando un componente de reconocimiento de caracteres.

En la parte superior de la ventana, hay una barra de estado con el título de la ventana y una serie de iconos de notificación, indicando parámetros como el volumen, la hora o la cantidad de batería disponible. A la derecha del todo, hay un icono que permite ocultar la ventana. Este icono es configurable con la propiedad *MinimizeBox* del formulario, y puede ser *x* u *ok*. En el primer caso, al pulsar sobre el botón la ventana se minimiza, pero la aplicación se sigue ejecutando en segundo plano. En el segundo caso se cierra la ventana, y finaliza la ejecución de la aplicación.

2) SmartPhone

Los dispositivos *smartphone* normalmente no disponen de pantalla táctil, por lo que la interacción con el usuario se realiza mediante los botones del teléfono. En la parte superior de la ventana, aparece también una barra de información con el título de la ventana y algunos iconos que indican, por ejemplo, el estado de la batería y la cobertura. Sin embargo, esta barra de información no tiene icono para cerrar la ventana; ésta puede cerrarse al pulsar la tecla de

inicio o la tecla de cancelar, aunque solamente se minimiza porque sigue ejecutándose en segundo plano. Para cerrar la aplicación, es necesario añadir una opción adicional en el programa, por ejemplo una opción de menú.

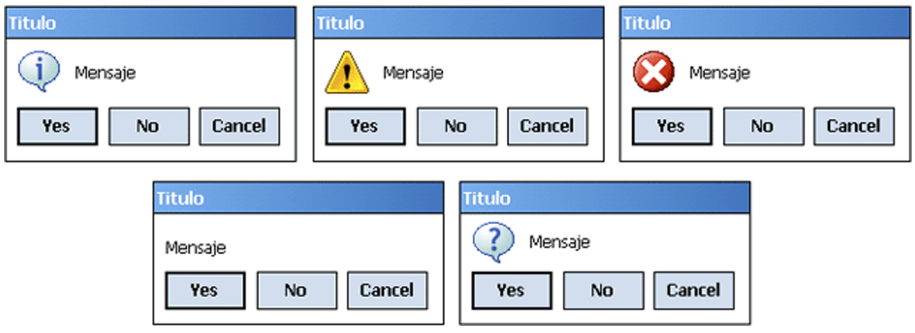
8.2.3. Cuadros de diálogo

Los cuadros de diálogo son también formularios, aunque se utilizan puntualmente para mostrar un mensaje al usuario o pedirle que seleccione o introduzca un determinado dato. Dentro del *namespace* de WinForms, existen algunos cuadros de diálogo predefinidos que permiten realizar tareas comunes.

Cuadros de diálogo predefinidos	
CommonDialog	Es una clase base a partir de la cual se pueden crear otros cuadros de diálogo.
FileDialog	Muestra un cuadro de diálogo en el que el usuario puede seleccionar un archivo. Es una clase abstracta que sirve de base para las clases *OpenFileDialog* y *SaveFileDialog*.
OpenFileDialog	Muestra un cuadro de diálogo en el que el usuario puede seleccionar un archivo para abrir.
SaveFileDialog	Muestra un cuadro de diálogo en el que el usuario puede seleccionar un archivo para grabar.
MessageBox	Muestra un mensaje al usuario y permite seleccionar una acción a realizar.

El aspecto del cuadro de diálogo *MessageBox* en *pocket PC* tiene el aspecto que se muestra en la siguiente figura. Los iconos mostrados corresponden con los valores de la enumeración *MessageBoxIcon* en orden (*Asterisk*, *Exclamation*, *Hand*, *None* y *Question*):

Figura 37. Cuadros de diálogo *MessageBox* en Pocket PC



8.2.4. Controles del .NET CF

La creación de una aplicación WinForms para dispositivos móviles se realiza de la misma forma que en WinForms normal, pero con un número de controles disponibles inferior. Además, muchos de los controles sólo están disponibles para los *pocket PC*, ya que los *smartphones*, al no tener pantalla táctil, tienen un reducido número de controles.

En la siguiente tabla, mostramos los controles principales y en qué tipos de dispositivos están disponibles:

Control	*Pocket PC*	*Smartphone*
Button	Disponible	No disponible
CheckBox	Disponible	Disponible
ComboBox	Disponible	Disponible
ContextMenu	Disponible	No disponible
DataGrid	Disponible	Disponible
DateTimePicker	Disponible	Disponible
HScrollBar	Disponible	Disponible
Label	Disponible	Disponible
ListBox	Disponible	No disponible
ListView	Disponible	Disponible
MainMenu	Disponible	Disponible
Panel	Disponible	Disponible
PictureBox	Disponible	Disponible
ProgressBar	Disponible	Disponible
RadioButton	Disponible	No disponible
StatusBar	Disponible	No disponible
TabControl	Disponible	No disponible
TextBox	Disponible	Disponible
Timer	Disponible	Disponible
ToolBar	Disponible	No disponible
TreeView	Disponible	Disponible
VScrollBar	Disponible	Disponible

Al diseñar una aplicación para Windows Mobile, es importante tener en cuenta una serie de normas, que están descritas en el programa de Microsoft "Designed for Windows Mobile". A continuación, mencionamos algunas de dichas normas:

- No utilizar fuentes de letra de tamaño fijo.

- Disponer de un único nivel de menús (sin submenús).

- Si la aplicación está diseñada para ser utilizada con el Stylus (el "lápiz" de la PDA), los botones deben ser de 21 × 21 píxeles. En cambio, si se han de pulsar con el dedo, la medida debe ser de 38 × 38 píxeles.

- Revisar que todo se vea bien, independientemente de la iluminación.

- Colocar los elementos que se tengan que pulsar, en la parte baja de la pantalla. De esta forma, la mano del usuario no tapará la pantalla.

- Reducir al máximo los datos que el usuario tenga que escribir mediante el teclado virtual (llamado SIP, del inglés *Software Input Panel*).

- No rellenar las listas con un número demasiado elevado de elementos.

- Todas las aplicaciones deben registrar un icono de 16 × 16 píxeles, y otro de 32 × 32 píxeles.

- La barra de navegación debe mostrar siempre el nombre de la aplicación, y ningún otro dato más.

- Toda aplicación debe disponer de un instalador que se encargue de crear un acceso directo al ejecutable. Además, deberá disponer de un desinstalador que lo deje todo tal y como estaba inicialmente.

8.2.5. Despliegue de aplicaciones

Al ejecutar una aplicación, Visual Studio la copia automáticamente al dispositivo o al emulador. Si éste no tiene instalada una versión compatible del .NET CF, Visual Studio la instala automáticamente.

Recomendación

Si podemos, es más recomendable trabajar con un dispositivo real que con el emulador.

La otra posibilidad es realizar una instalación manual, que consiste en copiar los ensamblados de la aplicación en una carpeta del dispositivo.

Si queremos distribuir la aplicación, será necesario ofrecer un programa de instalación. Para ello, crearemos un proyecto de tipo *Other project types*/*Setup and deployment/CAB Project*, que acabará generando un archivo CAB que instalará la aplicación de forma muy sencilla.

8.3. Aplicaciones web para dispositivos móviles

El desarrollo de aplicaciones web para dispositivos móviles es más complejo que para web normales por diversos motivos:

- Existen diferentes lenguajes soportados por unos u otros dispositivos, por ejemplo, HTML, WML o cHTML.

- Los dispositivos pueden tener pantallas muy variadas: distintos tamaños, distintos números de filas y columnas de texto, orientación horizontal o vertical, pantallas en color o en blanco y negro, etc.

- Las velocidades de conexión pueden ser muy diferentes entre dispositivos: GPRS (40 kbps), EDGE (473 kbps) o HSDPA (14 Mbps).

cHTML

cHTML (Compact HTML) es un subconjunto de HTML utilizado en los teléfonos móviles DoCoMo de Japón.

Para solventar estas diferencias, Visual Studio 2005 ofrecía soporte al desarrollo de aplicaciones ASP.NET específicas para dispositivos móviles, mediante los controles ASP.NET mobile. Cuando un dispositivo se conecta a una página ASP.NET mobile, el servidor recupera información acerca del hardware del dispositivo, su navegador y la velocidad de conexión. Basándose en esta información, los controles ASP.NET mobile producen una u otra salida en función del lenguaje de marcado utilizado, las capacidades del navegador, las propiedades de la pantalla y la velocidad de conexión.

¡Atención!

Visual Studio 2008 actualmente no ofrece soporte específico al desarrollo de aplicaciones ASP.NET para móviles.

De esta forma, en Visual Studio 2005, para crear una aplicación con páginas ASP.NET para móviles, hay que crear un sitio web *ASP.NET* y añadir formularios *Mobile Web Forms*. Una vez hecho esto, se puede comprobar cómo la página ASP.NET hereda de MobilePage, y la paleta de controles muestra únicamente los controles disponibles para móviles.

Recientemente, los dispositivos móviles han evolucionado mucho, hasta el punto de que, hoy en día, existen diversos modelos que permiten navegar sin problemas por páginas HTML "normales". En consecuencia, y debido al gran esfuerzo que supone el desarrollo de los controles ASP.NET Mobile, parece probable pensar que Microsoft acabe descontinuando esta línea de trabajo, para centrarse en las páginas ASP.NET "normales", que en breve serán accesibles por la mayoría de dispositivos móviles del mercado.

Esfuerzo de desarrolo

Es necesario asegurar la compatibilidad con infinidad de móviles diferentes, lo que hace el desarrollo muy tedioso.

Por otra parte, merece la pena comentar que Silverlight ya está disponible para Symbian S60, y en breve estará disponible para Windows Mobile.

Bibliografía

Gañán, David (2008). Material del máster de *.NET* de la UOC.

Gibbs, M.; Wahlin, D. (2007). *Professional ASP.NET 2.0 AJAX*. Wiley Publishing. Inc.

Johnson, G.; Northrup, T. (2006). *Microsoft .NET Framework 2.0 Web-Based Client Development*. Microsoft Press.

Johnson, B.; Madziak, P.; Morgan, S. (2008). *NET Framework 3.5 Windows Communication Foundation*. Microsoft Press.

Northrup, T.; Wildermuth, S. (2008). *Microsoft .NET Framework Application Development Foundation* (2.ª ed.). Microsoft Press.

Stoecker, M. A.; Stein, S. J.; Northrup, T. (2006). *Microsoft .NET Framework 2.0 Windows-Based Client Development*. Microsoft Press.

Stoecker, M. A. (2008). *Microsoft .NET Framework 3.5 Windows Presentation Foundation*. Microsoft Press.

Wigley, A.; Moth, D.; Foot, P. (2007). *Microsoft Mobile Development Handbook*. Microsoft Press.