

Ejercicios
con soluciones

Desarrollo de aplicaciones

C# con

Microsoft

Visual Studio

.NET

Curso práctico

Proyectos
completos

Borja Orbegozo Arana

**Desarrollo de
aplicaciones**

C#

con *Visual Studio*

.NET

Curso práctico

Borja Orbegozo Arana

ΔΔ Alfaomega

 **Altaria**
publicaciones

Revisado por:
Beatriz Vega y Carlos Martínez Peña

Datos catalográficos

Orbegozo, Borja
Desarrollo de aplicaciones C# con Visual Studio .NET
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México

ISBN: 978-607-622-220-1

Formato: 17 x 23 cm

Páginas: 284

Desarrollo de aplicaciones C# con Visual Studio .NET

Borja Orbegozo Arana

ISBN: 978-84-943007-0-7, edición en español publicada por Publicaciones Altaria S.L.,
Tarragona, España

Derechos reservados © PUBLICACIONES ALTARIA, S.L.

Primera edición: Alfaomega Grupo Editor, México, junio 2015

© 2015 Alfaomega Grupo Editor, S.A. de C.V.
Pitágoras 1139, Col. Del Valle, 03100, México D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: <http://www.alfaomega.com.mx>
E-mail: atencionalcliente@alfaomega.com.mx

ISBN: 978-607-622-220-1

Derechos reservados:

Esta obra es propiedad intelectual de su autor y los derechos de publicación en lengua española han sido legalmente transferidos al editor. Prohibida su reproducción parcial o total por cualquier medio sin permiso por escrito del propietario de los derechos del copyright.

Nota importante:

La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro, ni por la utilización indebida que pudiera dársele.

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Pitágoras 1139, Col. Del Valle, México, D.F. – C.P. 03100.
Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia,
Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24. Santiago. Chile

A toda mi familia y a mi pareja.

¿A quién va dirigido este libro?

El libro va dirigido a cualquier usuario con conocimientos básicos o nulos sobre programación y que desee ampliarlos o adquirirlos, respectivamente. También va dirigido a quienes deseen aprender a diseñar aplicaciones de escritorio a partir de los problemas reales que se deseen solucionar.

Será un manual imprescindible en academias y centros de formación, así como para usuarios autodidactas.

Convenciones generales

El contenido del libro es lo más didáctico posible para que el lector pueda hacer pruebas en su propio ordenador mientras progresiona en la lectura. Nuestro objetivo es que el usuario tenga en sus manos un manual sencillo de seguir y con mucha parte práctica, de manera que sirva como herramienta tanto para fines académicos como productivos.

El libro contiene muchas imágenes y mucho código fuente para facilitar más la lectura y el aprendizaje, haciéndolo más ameno.

Al final del libro se encuentran los ejemplos prácticos más importantes y las soluciones a todos los ejercicios planteados en los diferentes capítulos.

Índice general

¿A quién va dirigido este libro?	5
Convenciones generales	5

Capítulo 1

Introducción.....	11
<i>Visual Studio .NET</i>	11
El entorno <i>.NET Framework</i>	12
El entorno de desarrollo <i>Visual Studio .NET</i>	12
Gestión y tipos de proyectos	14
Bibliotecas.....	17
Biblioteca de clases de <i>.NET Framework</i>	17
Uso de una biblioteca de clases en las aplicaciones	18
Instaladores de aplicaciones	19

Capítulo 2

Programación orientada a objetos	21
Introducción.....	21
Clases y objetos.....	22
Encapsulamiento	24
Herencia.....	24
Polimorfismo	26
Ejercicios.....	27
Ejercicio 2.1.....	27
Ejercicio 2.2.....	28
Ejercicio 2.3.....	28

Capítulo 3

Programación base.....	29
Bases sintácticas de C#	29
Espacios de trabajo.....	31
Clases	33
Variables y constantes	34
Operadores.....	37
Métodos	42
Constructores.....	46
Campos	49
Propiedades	51
Control de flujo.....	52

La instrucción <i>switch</i>	55
Estructuras iterativas	58
El bucle <i>for</i>	58
El bucle <i>while</i>	61
El bucle <i>do</i>	62
Rupturas de código.....	63
Recursividad	65
<i>Arrays</i>	66
Conversiones de tipos	70
Interfaces	72
Práctica paso a paso	72
Ejercicios.....	75
Ejercicio 3.1.....	75
Ejercicio 3.2.....	75

Capítulo 4

Programación gráfica..... 77

Programación gráfica.....	77
Programación orientada a eventos	77
Formularios.....	78
Herramientas	81
Controles comunes.....	82
Contenedores	107
Menús y barras de herramientas	116
Cuadros de diálogo.....	125
Acceso a bases de datos.....	135
Práctica paso a paso	156
Ejercicios.....	182
Ejercicio 4.1.....	182
Ejercicio 4.2.....	183
Ejercicio 4.3.....	184
Ejercicio 4.4.....	185

Capítulo 5

Proyectos completos de programación.. 187

Proyecto completo con formularios	187
Proyecto completo con bibliotecas	238

Capítulo 6

Instalación de aplicaciones	249
Las aplicaciones <i>ClickOnce</i>	249
Publicación de proyectos	250
Instalación de proyectos.....	253

Capítulo 7

Soluciones a los ejercicios	257
Ejercicio 2.1.....	257
Ejercicio 2.2.....	258
Ejercicio 2.3.....	258
Ejercicio 3.1.....	259
Ejercicio 3.2.....	260
Ejercicio 4.1.....	260
Ejercicio 4.2.....	268
Ejercicio 4.3.....	271
Ejercicio 4.4.....	275

Introducción

CAPÍTULO 1

Visual Studio .NET

Visual Studio .NET es un conjunto de herramientas de desarrollo que permiten construir diferentes tipos de aplicaciones tanto para el escritorio como para la web o el móvil. Dentro de *Visual Studio* se pueden desarrollar aplicaciones utilizando diferentes lenguajes de programación como *Visual Basic .NET*, *Visual C++ .NET*, *Visual C# .NET* y *Visual J# .NET*.

Se usa el mismo entorno de desarrollo integrado o *IDE* para cualquiera de estos lenguajes, lo que permite compartir herramientas y además facilita la creación de soluciones utilizando diferentes lenguajes.

Además, *.NET Framework* contiene una serie de funciones que pueden ser aprovechadas al programar y que ofrecen acceso a tecnologías que simplifican el desarrollo de los diferentes tipos de aplicaciones.

La versión de desarrollo que se va a utilizar en este manual es *Visual Studio .NET 2013*.

El entorno .NET Framework

.NET Framework es un entorno multilenguaje que consta de tres partes principales:

- *Common Language Runtime*: Este motor de tiempo de ejecución entra en funcionamiento tanto a la hora de ejecutar aplicaciones como durante el desarrollo de las mismas. En el momento de la ejecución, esta parte es la encargada de gestionar la memoria, de iniciar y parar procesos y subprocesos, de controlar las directivas de seguridad y de controlar las dependencias entre componentes. En el momento del desarrollo, esta parte simplifica el trabajo del programador reduciendo al máximo el código que debe escribir para que la lógica del programa se aproveche de los componentes reutilizables.
- Clases de programación unificadas: Existe un conjunto unificado, orientado a objetos, jerárquico y extensible a bibliotecas de clases que pueden ser utilizadas por los programadores. Se puede tener acceso a las mismas bibliotecas desde cualquier lenguaje de programación. Se permiten la herencia, el control de errores y la depuración de errores entre diferentes lenguajes, accediendo desde cada uno de ellos de manera similar.
- *ASP .NET*: Está construido sobre las clases de programación de .NET Framework y permite tener un modelo con un conjunto de controles que encapsulan elementos comunes de las interfaces HTML para desarrollar aplicaciones web. Los controles se ejecutan en los servidores y el modelo de programación que tienen está orientado a objetos.

El entorno de desarrollo Visual Studio .NET

Cuando se arranca la aplicación por primera vez, hay que definir para qué lenguaje se prefiere generar dicho entorno. El programador debe decidir cuál es el lenguaje que va a utilizar con más frecuencia. Para el seguimiento de este manual se sugiere utilizar la configuración de desarrollo para C#.

En esta versión 2013 también se puede elegir un tema de color, que va a ser el que se va a utilizar en el entorno de desarrollo.

1. Introducción

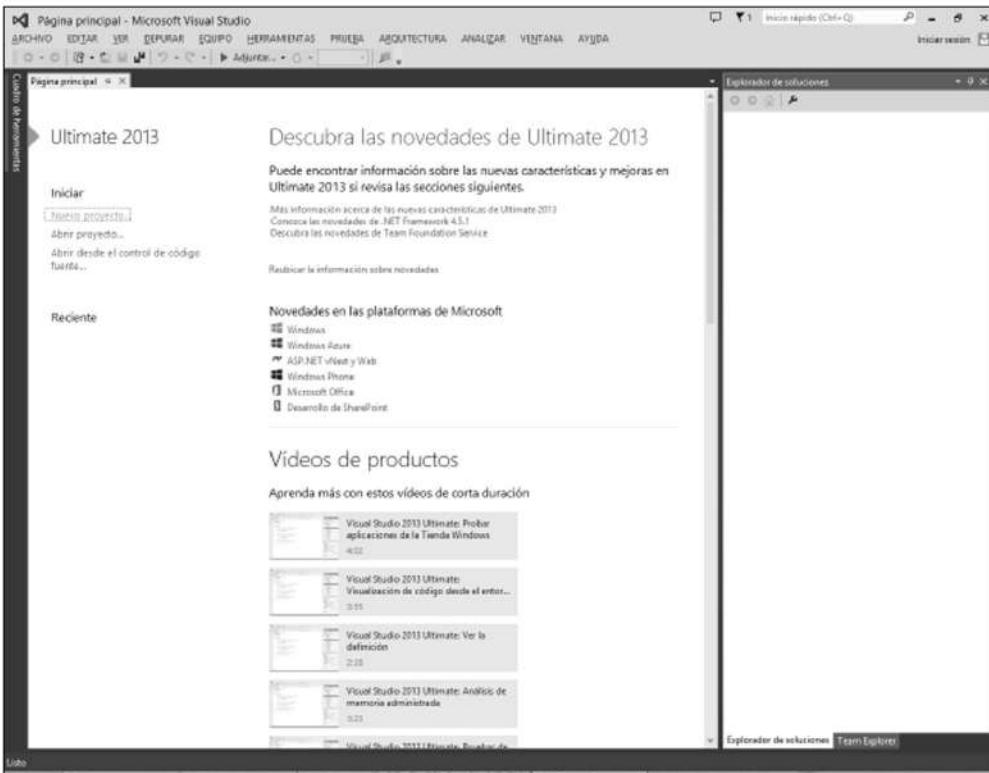
La selección de estos parámetros se puede ver en la siguiente imagen.



Una vez que se ha definido la configuración deseada, se debe pulsar el botón *Iniciar Visual Studio* y se llega a la pantalla inicial, desde la que

Desarrollo de aplicaciones C# con Visual Studio .NET

se pueden iniciar los nuevos proyectos o escoger proyectos ya creados anteriormente. El aspecto de dicha ventana es el de la siguiente imagen.

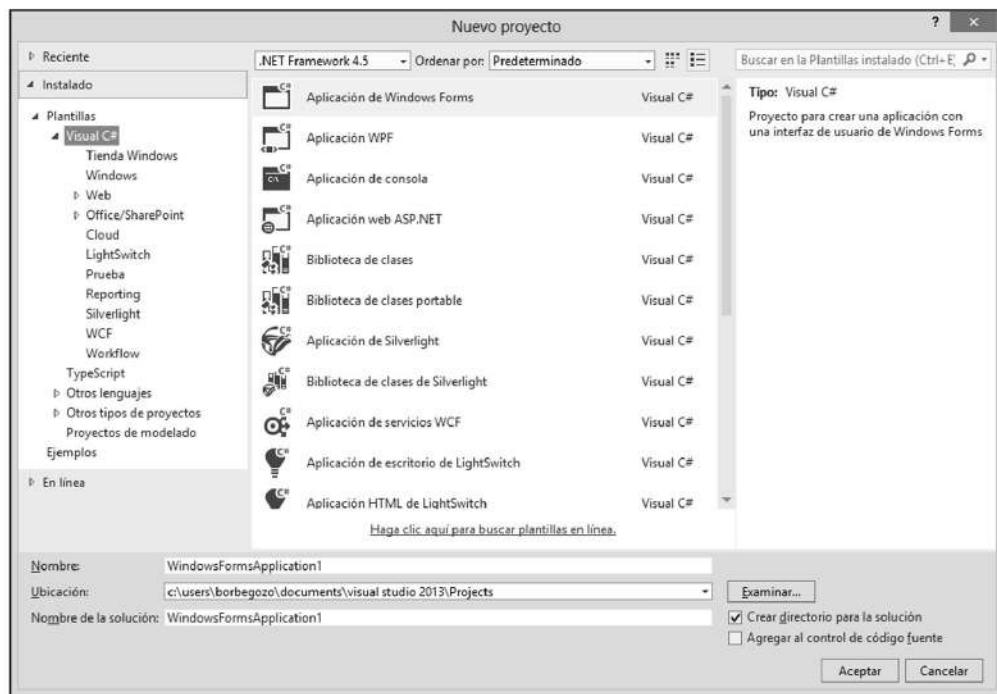


Gestión y tipos de proyectos

Las aplicaciones que se crean utilizando *Visual C#* están englobadas en proyectos. Los proyectos contienen la información de todos los componentes que engloban el conjunto de la solución. Dichos componentes pueden ser formularios, clases, bibliotecas, informes...

Cuando se entra en el entorno de desarrollo para crear un nuevo proyecto, se abre una ventana en la que se solicita la ubicación en la que se quiere guardar el proyecto, el nombre del mismo, el nombre de la solución y, lo más importante, el tipo de proyecto que se quiere desarrollar. Dicha ventana es la siguiente:

1. Introducción



Los tipos de proyectos que se pueden desarrollar son los siguientes:

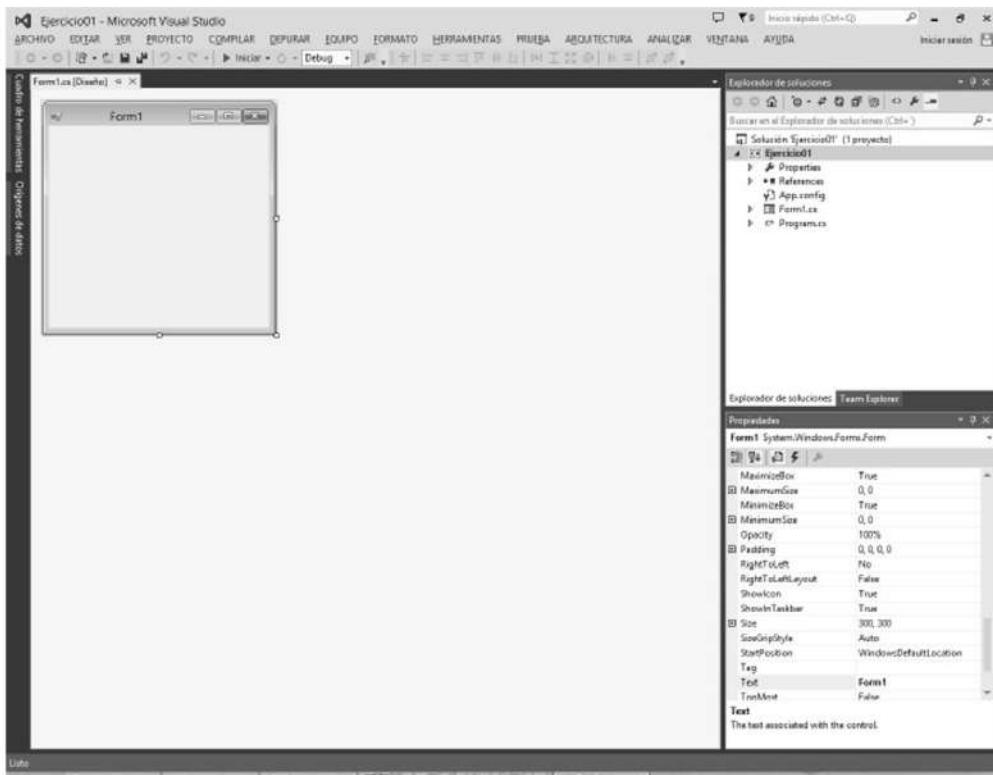
- Aplicación de *Windows Forms* (para crear una aplicación con una interfaz de usuario de *Windows Forms*).
- Aplicación *WPF* (para crear una aplicación cliente de *Windows Presentation Foundation*).
- Aplicación de consola (para crear una aplicación de línea de comandos).
- Aplicación web *ASP .NET* (es una plantilla de proyecto para crear aplicaciones *ASP .NET*, como *ASP .NET Web Forms*, *MVC* o *Web API*).
- Biblioteca de clases (para crear una biblioteca de clases de *C#* generando una *DLL*).
- Biblioteca de clases portable (para crear una biblioteca de clases de *C#* generando una *DLL* que puede ejecutarse en *Windows*, *Silverlight* y *Windows Phone*).

- Aplicación de *Silverlight* (para crear aplicaciones para Internet mediante *Silverlight*).
- Biblioteca de clases de *Silverlight* (para crear una biblioteca de clases de *Silverlight*).
- Aplicación de servicios WCF (para crear una aplicación de servicio WCF que se hospede en *IIS/WAS*).
- Aplicación de escritorio de *LightSwitch* (para crear una aplicación de línea de negocio con *Siverlight*).
- Aplicación HTML de *LightSwitch* (se aprovecha de C#, HTML, JavaScript, jQuery y jQuery Mobile para crear aplicaciones de móviles enriquecidas).
- Libro de *Excel 2013* (para crear extensiones de código administrado subyacentes de un libro de *Excel 2013* nuevo o existente).
- Complemento de *Outlook 2013* (para crear un complemento de código administrado para *Outlook 2013*).
- Documento de *Word 2013* (para crear extensiones de código administrado subyacentes de un documento de *Word 2013* nuevo o existente).
- Biblioteca de actividad (para crear una biblioteca de actividad de flujo de trabajo).
- Aplicación de servicio de flujo de trabajo WCF (para crear una aplicación de servicio de flujo de trabajo WCF que se hospede en *IIS/WAS* y sea administrada por *Microsoft AppFabric* para *Windows Server*).

Una vez que se ha elegido el tipo de proyecto que se quiere desarrollar, se llega a la ventana principal para el desarrollo de la aplicación. En función del tipo de proyecto, la ventana se adaptará para su correspondiente desarrollo.

Por ejemplo, si se desea crear un proyecto que sea una aplicación de *Windows Forms*, se llegará a un entorno con este aspecto:

1. Introducción



Bibliotecas

Las bibliotecas de clases son una parte fundamental dentro del desarrollo de aplicaciones en *Visual C#*. Por un lado, el propio entorno de desarrollo utiliza la biblioteca de clases de *.NET Framework* para todo tipo de actividades y, por otro lado, el desarrollador puede generar sus propias bibliotecas.

Biblioteca de clases de .NET Framework

La mayoría de los proyectos de *Visual C#* utilizan esta biblioteca para diferentes actividades, como el acceso al sistema de archivos, la manipulación de cadenas, los controles de las interfaces de usuario, etc.

La biblioteca de clases se organiza en espacios de nombres que contienen clases y estructuras relacionadas entre ellas.

.NET Framework contiene clases, interfaces y tipos de valores que agilizan y optimizan el proceso de desarrollo. Además, proporcionan acceso a las funciones del sistema.

Los tipos que se incluyen realizan las siguientes funciones:

- Representar tipos de datos básicos y excepciones.
- Encapsular estructuras de datos.
- Realizar acciones de E/S.
- Obtener acceso a información sobre tipos cargados (se acepta la sobrecarga de métodos).
- Llamar a las comprobaciones de seguridad.
- Proporcionar acceso a datos.
- Proporcionar una interfaz gráfica para el usuario.

Los tipos usan un esquema de nomenclatura con sintaxis de punto, lo que denota que existe una jerarquía. El esquema de nomenclatura que tiene facilita a los desarrolladores su tarea, pudiendo crear grupos jerárquicos de tipos y asignarles nombres de forma coherente. También permite identificar únicamente los tipos mediante su nombre completo.

Por ejemplo, el espacio de nombres *System* contiene los nombres de la raíz de los tipos fundamentales de .NET Framework y dentro hay clases que representan todos los tipos de datos base.

Uso de una biblioteca de clases en las aplicaciones

Para poder utilizar las clases de una biblioteca es necesario añadir al programa una directiva *using* con el espacio de nombres que se quiere usar. De esta forma, se agregan automáticamente las referencias a las *DLL* utilizadas. En el proyecto se podrán ver en todo momento las referencias que han sido agregadas.

Dentro de la aplicación se pueden crear instancias a los tipos que estén contenidos en la biblioteca, realizar llamadas a métodos y responder a eventos que se hayan declarado en ella, de la misma forma que si estuvieran en el propio código fuente.

Instaladores de aplicaciones

En *Visual C#*, de la misma forma que en el resto de lenguajes del entorno de desarrollo *Visual Studio 2013*, se tiene la posibilidad de publicar las aplicaciones desarrolladas generando un instalador para la aplicación. Así como en versiones anteriores había que crear un tipo de proyecto específico para las instalaciones, en esta versión es más sencillo y directamente se puede generar todo desde el propio proyecto desarrollado.

Las posibilidades que se ofrecen a la hora de elegir un destino para los archivos del instalador son las tres siguientes:

- Alojar en web.
- Alojar en un recurso compartido de archivos.
- Alojar en un CD-ROM o DVD-ROM.

En el apartado 6 de este libro se explica con más detalle todo lo que lleva el proceso de generación y testeо de un instalador para aplicaciones.

Programación orientada a objetos

CAPÍTULO
2

Introducción

La programación orientada a objetos consiste en un modo de trabajo que permite centrarse en la solución al problema de forma natural. Permite abstraerse de muchas tareas, como las relacionadas con el diseño de la pantalla o la interacción con el usuario, para escribir el código que realmente es necesario para resolver el problema.

Es importante definir bien los objetos y también las acciones relacionadas con ellos. A partir de ahí, la programación se simplifica en gran medida. Por ejemplo, si se quiere utilizar una base de datos desde la programación orientada a objetos, se tendrán acciones como guardar los datos, leerlos, buscar un dato o grupo de ellos, etc. Después, en el código, simplemente se llenarán las propiedades del objeto con los datos correspondientes y se llamará a la acción deseada, de manera que el desarrollo queda mucho más limpio y legible.

Otra gran ventaja es que, si se modifican los contenidos de las acciones, ello será independiente del resto del código. Es decir, que la buena programación no deberá modificarse.

Clases y objetos

Las clases y los objetos están muy relacionados, pero no son lo mismo. Primero se crean las clases, que son las que definen cómo van a ser después los objetos. Un objeto lo que hace es instanciar una clase y con ello ya está definida la estructura que va a tener dicho objeto. Para crear un objeto previamente tienen que haberse definido las clases.

Por ejemplo, se puede crear una clase para guardar las características de las personas que permita además realizar una serie de acciones. Después se crea el objeto que define a cada persona en particular.

```
class Persona
{
    public Persona(string ojos, string altura, string peso,
                   string sexo)
    {
        this.Ojos=ojos;
        this.Altura=altura;
        this.Peso=peso;
        this.Sexo=sexo;
    }

    public string Ojos;
    public string Altura;
    public string Peso;
    public string Sexo;

    public void Hablar(string texto)
    {
        // Se muestra la conversación a decir
        Console.WriteLine("Voy a decir: " + texto);
    }

    public void Comer(double calorias)
    {
        // Se indican las calorías que se ganan al comer
        Console.WriteLine("Ganando " + cantidad + " calorías");
        this.peso += calorías/1000;
    }

    public void Correr(double calorias)
    {
        // Se indican las calorías que se pierden al correr
        Console.WriteLine("Perdiendo " + cantidad + " calorías");
        this.peso -= calorías/1000;
    }
}
```

2. Programación orientada a objetos

Para utilizar dicha clase desde una aplicación, se utiliza el método *Main* y en él se incluyen las asignaciones y las llamadas a las acciones.

```
class EjemploPersonaApp
{
    static void Main()
    {
        Persona Personal1=new Persona("Verdes", "187", "85", "V");

        Console.WriteLine("Descripción de la persona:");
        Console.WriteLine("Ojos: ", Personal1.Ojos);
        Console.WriteLine("Altura: ", Personal1.Altura);
        Console.WriteLine("Peso: ", Personal1.Peso);
        Console.WriteLine("Sexo: ", Personal1.Sexo);

        Personal1.Hablar("Me llamo Borja");
        Console.WriteLine("Color de mis ojos: ", Personal1.Ojos);
        Personal1.Correr(1000);
        Console.WriteLine("Mi peso actual es: ", Personal1.Peso);
    }
}
```

Al ejecutar el código de esta aplicación, el resultado que se visualiza por la consola es el siguiente:

```
Descripción de la persona:
Ojos: Verdes
Altura: 187
Peso: 85
Sexo: V
Me llamo Borja
Color de mis ojos: Verdes
Mi peso actual es: 84
Tras nadar peso:85
```

Aunque por el momento el código no se entienda todavía, lo importante es que se vea claro lo que son datos y lo que son acciones. En este caso, los datos son:

- *Ojos*
- *Altura*
- *Peso*
- *Sexo*

Las acciones del objeto son:

- *Hablar*
- *Correr*

Cuando se define el objeto *Personal1*, no hay que definir nada, ya que todo está previamente diseñado en la clase *Persona*.

Encapsulamiento

El encapsulamiento es la capacidad que tienen los objetos tanto para proteger sus datos como para ocultar su propio código. Los objetos deben ser accesibles a través de la interfaz que los define y que garantiza que el uso del objeto es el adecuado desde la parte de desarrollo de la aplicación.

En el ejemplo del punto anterior, al desarrollador no le importa cuál es el contenido de las acciones *Hablar* y *Correr*. Ni siquiera en qué lenguaje están hechas, puesto que podrían formar parte de una biblioteca desarrollada en otro lenguaje. Lo único que le interesa al desarrollador es saber el resultado al realizar las acciones, pero no su definición ni su contenido.

Respecto a la protección de datos, en el ejemplo anterior se ve que la propiedad *Peso* no se puede modificar directamente, pero al llamar a las acciones *Comer* y *Correr*, dicha propiedad se ve modificada. En función de la cantidad de calorías que se le pase como parámetro, la acción *Comer* va a incrementar la cantidad del peso y la acción *Correr* la va a reducir. Esta característica permite que el dato se modifique de la manera adecuada y no arbitrariamente.

Otra de las ventajas que tiene la protección de datos es que el código es portable, es decir, que se puede usar la misma clase en diferentes programas sin necesidad de volver a escribirla.

Herencia

Otra de las bases importantes dentro de la programación orientada a objetos es la herencia. Gracias a esta característica se pueden definir clases nuevas que estén basadas en otras clases ya creadas.

Siguiendo con el mismo ejemplo de los puntos anteriores, se va a crear una nueva clase que esté basada en la clase *Persona*, pero a la que se le añadirá una nueva característica. A la nueva persona se le va a añadir la capacidad de nadar, lo cual se hará definiendo la nueva clase de la siguiente manera:

2. Programación orientada a objetos

```
class PersonaNadadora:Persona
{
    public PersonaNadadora(string ojos, string altura, string peso,
    string sexo): base(ojos, altura, peso, sexo) {}

    public void Nadar()
    {
        // La práctica de la natación hará volver al peso inicial
        Console.WriteLine("Nadando para volver al peso ideal");
        this.peso = 85;
    }
}
```

Lo que hay que tener en cuenta es que todas las acciones definidas en la clase *Persona*, también se pueden utilizar en la nueva clase *PersonaNadadora*, sin necesidad de volver a definirlas. Todas esas acciones han sido heredadas por la nueva clase.

La manera de utilizar esta nueva clase, tanto con la nueva acción como con las anteriores, va a ser exactamente igual que con la clase principal.

```
class EjemploPersonaApp
{
    static void Main()
    {
        PersonaNadadora Persona1=new PersonaNadadora ("Verdes",
        "187", "85", "V");

        Console.WriteLine("Descripción de la persona:");
        Console.WriteLine("Ojos: «, Persona1.Ojos");
        Console.WriteLine("Altura: «, Persona1.Altura");
        Console.WriteLine("Peso: «, Persona1.Peso");
        Console.WriteLine("Sexo: «, Persona1.Sexo");

        Persona1.Hablar("Me llamo Borja");
        Console.WriteLine("Color de mis ojos: «, Persona1.Ojos");
        Persona1.Correr(1000);
        Console.WriteLine("Mi peso actual es: «, Persona1.Peso");
        Persona1.Nadar();
        Console.WriteLine("Tras nadar peso: «, Persona1.Peso");
    }
}
```

Al ejecutar el código de esta aplicación, el resultado que se visualiza en la consola es el siguiente:

```
Descripción de 1a persona:  
Ojos: Verdes  
Altura: 187  
Peso: 85  
Sexo: V  
Me llamo Borja  
Color de mis ojos: Verdes  
Mi peso actual es: 84  
Tras nadar peso: 85
```

Como se puede apreciar, ahora el objeto *Personal1* tiene todo idéntico a la anterior, pero se le ha añadido la acción *Nadar*, implementada por la clase heredada *PersonaNadadora*.

Siguiendo esta misma filosofía, se pueden ir construyendo clases cada vez más complejas a partir de otras clases más sencillas. Eso sí, hay que tener en cuenta que *Visual C#* solamente soporta herencia simple y no herencia múltiple. Eso quiere decir que a partir de una clase se puede construir otra, pero no a partir de varias clases.

Polimorfismo

La otra característica que aporta la programación orientada a objetos es el polimorfismo, que permite la reescritura de las acciones de la clase original, de manera que ciertas instancias en particular puedan tener comportamientos diferentes.

Para utilizar el polimorfismo, los métodos deben estar definidos como virtuales para que podamos modificarlos. Así, uno de los métodos del ejemplo se podría haber definido de esta manera:

```
Public virtual void Comer(double calorias)  
{  
    // Se indican las calorías que se ganan al comer  
    Console.WriteLine("Ganando " + cantidad + " calorías");  
    this.peso += calorías/1000;  
}
```

2. Programación orientada a objetos

Todos aquellos métodos que lleven la palabra clave *virtual* se van a poder reescribir. Por ejemplo, una clase que utilice el polimorfismo podría ser la siguiente:

```
class PersonaGlotona : Persona
{
    public PersonaGlotona(string ojos, string altura,
        string peso, string sexo) : base(ojos, altura, peso,
        sexo) {}

    Public override void Comer(double calorías)
    {
        // Se indican las calorías que se ganan al comer
        Console.WriteLine(`Ganando « + 2*cantidad + " calorías");
        this.peso += 2*calorías/1000;
    }
}
```

Como se puede apreciar, lo único que se modifica es la clase heredada, la clase original permanece intacta. Y como se ve, lo único que se sobrescribe es aquello de la clase base que no sirve, como es en este caso la acción *Comer*. Por supuesto, el compilador va a ejecutar siempre el método sobrescrito.

Cuando se crea el objeto se puede indicar que es de la clase original, pero utilizar la clase derivada después al realizar la asignación. En este caso, el método que se va a ejecutar también va a ser el método sobrescrito.

Ejercicios

Ejercicio 2.1

Crea una clase principal que sirva para guardar los datos de un coche genérico y que permita guardar en los datos las siguientes características:

- Marca
- Modelo
- Velocidad máxima

Además debe tener un par de acciones que hagan lo siguiente:

- Acelera (incrementa la velocidad de 5 en 5).
- Decelera (reduce la velocidad de 5 en 5).

Ejercicio 2.2

Crea una clase secundaria heredada a partir de la clase del ejercicio anterior. En dicha clase, además, se añadirá otra nueva acción:

- Frena (deja la velocidad a 0).

Ejercicio 2.3

Crea una clase principal que sirva para guardar los datos de un ordenador y que permita guardar en estos datos las siguientes características:

- Marca
- Procesador
- Memoria
- Disco duro

Además debe tener un par de acciones que hagan lo siguiente:

- MayorCapacidad (incrementa la capacidad del disco duro de 100 en 100 Gb).
- MenorCapacidad (reduce la capacidad del disco duro de 100 en 100 Gb).

Programación base

CAPÍTULO **3**

Bases sintácticas de C#

La sintaxis de C# es muy similar a la de otros lenguajes como C, C++ o Java. Lo que hay que tener en cuenta es que se han adaptado estos lenguajes para que tengan una sintaxis más similar a los demás lenguajes del entorno de desarrollo que usa .NET Framework. La razón de ello es que este Framework está compuesto por el *Common Language Runtime* y por la biblioteca de clases correspondiente. Este entorno de ejecución permite que, al compilar desde cualquiera de los lenguajes, el código sea el mismo, de forma que después se puedan reutilizar elementos de un lenguaje desde otro.

Por supuesto, el editor de *Visual C#* va a ayudar mucho a la hora de escribir el código, puesto que cuando se detectan errores, las palabras aparecen subrayadas en rojo o en azul dependiendo del tipo de error.

La declaración de variables es exactamente igual a la de C. Por ejemplo, se pueden declarar dos variables de tipo entero o de tipo puntero a entero de la siguiente forma:

```
int a;  
int* pA;
```

Lo que hay que tener en cuenta es que si bien sintácticamente se declara igual en ambos lenguajes, el significado no es el mismo. Lo que antes era un tipo entero sin más, en C# pasa a ser un objeto de la clase *System.Int32*, puesto que excepto los punteros, todo lo demás son objetos.

Otro tema importante es que todas las declaraciones y todas las instrucciones deben terminar con un punto y coma. La ventaja que presenta indicar el final de la instrucción, es que permite que dicha instrucción ocupe varias líneas sin necesidad de indicar nada especial en los saltos de línea.

Por otra parte, los bloques de código estarán encerrados entre llaves de apertura (*{*) y cierre (*}*). Los bloques de código pueden estar comprendidos dentro de las funciones o métodos, pero también son bloques los que están contenidos dentro de las instrucciones condicionales o iterativas.

Por ejemplo, la siguiente función tiene un bloque de código que a su vez está dentro de otro bloque de código.

```
public bool Negativo(int numero)
{
    if (numero<0)
    {
        return true;
    }
    return false;
}
```

Como se puede apreciar en el ejemplo, los bloques de código comienzan y terminan de la misma manera, con lo que se puede perder algo de legibilidad cuando se tienen varios niveles de bloques. Para ello, es tarea importante del desarrollador ir creando el código de manera muy estructurada y ordenada. Hay que tener en cuenta también que antes de las llaves de apertura no se debe poner el punto y coma, puesto que la instrucción no termina hasta que llega la llave de cierre.

El entorno de desarrollo ayuda a la hora de colocar las tabulaciones y va generando las llaves en su lugar correspondiente, pero no impide que después el desarrollador lo modifique. Se aconseja no modificar dichas tabulaciones y dejarlas tal y como el entorno propone.

Los programas que se escriben en este lenguaje están organizados en clases y estructuras. Todo el código va a estar siempre incluido en una clase global o en una estructura. Ello está ligado a que las funciones sean en realidad métodos, puesto que siempre van a estar ligados a una clase.

3. Programación base

De la misma forma, las variables y constantes van a ser propiedades de la clase.

Todas las aplicaciones van a tener un método principal que se debe llamar *Main* y que tiene que ser público y estático. Lo normal, o lo más recomendable, es que dicho método se encuentre dentro de la clase principal con el nombre de la aplicación.

Por supuesto, se debe tener en cuenta que en este lenguaje se distinguen las mayúsculas de las minúsculas. No es lo mismo la variable *a* que la variable *A*, ni es lo mismo el método *conectar* que el método *Conectar*. En cualquier caso, el entorno de desarrollo va a ayudar marcando como incorrectos aquellos casos en los que el desarrollador se equivoque en las mayúsculas y minúsculas. Puede darse un problema cuando existan ambas y se esté utilizando la equivocada.

Otra opción muy interesante es la sobrecarga de métodos. La sobrecarga se da cuando un método está definido varias veces con el mismo nombre, pero con diferentes parámetros. Ello está permitido y después, en las llamadas, se van a distinguir por la cantidad y tipo de los parámetros.

Importante también es saber comentar el código. Para ello se pueden usar dos opciones. Si se utiliza una doble barra (//), se puede poner un comentario que no ocupe más de una línea. Si se quiere poner un comentario largo de varias líneas, al comienzo se pondrá /* y al final se pondrá */. Por ejemplo:

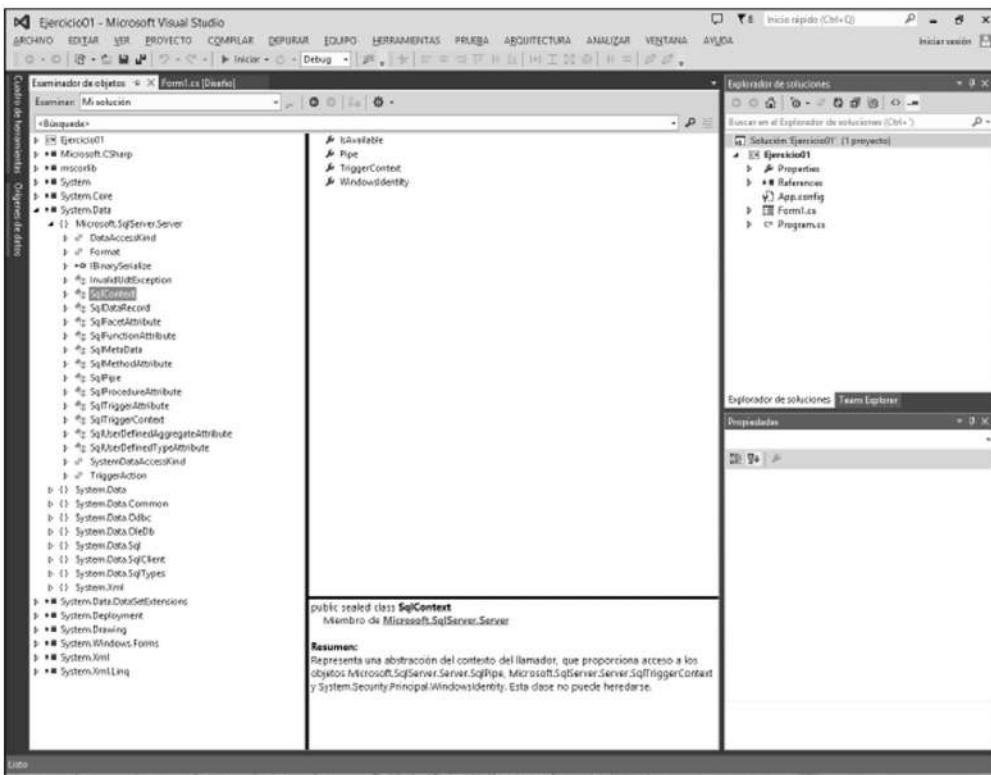
```
// Comentario que no ocupa más de una línea
/*
    Comentario que ocupa:
    una línea
    dos líneas
    tres líneas
    cuatro líneas
    cinco líneas
*/
```

Espacios de trabajo

Los espacios de trabajo o espacios de nombres permiten organizar de manera sencilla todas las clases del programa, ya sean desarrollos propios o provenientes de *.NET Framework*.

Desarrollo de aplicaciones C# con Visual Studio .NET

Podemos ver un ejemplo de cómo se visualizan estos espacios de trabajo entrando en el *Examinador de objetos*.



En la parte izquierda, aparece la biblioteca de clases de *.NET Framework*, organizada en forma de árbol de una manera totalmente estructurada. Cuando se vayan generando las clases propias en desarrollo, aparecerán también en este espacio de trabajo, dentro de la misma estructura.

Lo que se recomienda es que los espacios de nombres comiencen con el nombre de la compañía, de manera que aunque después los nombres de clases puedan coincidir entre varias empresas, las bibliotecas se puedan utilizar sin que haya problema, puesto que el espacio de nombres será siempre diferente.

3. Programación base

Los espacios de trabajo, que pueden ser varios en un mismo proyecto, se definen de la siguiente manera:

```
namespace MiEspacio
{
    // Dentro irán las clases del espacio de trabajo
}
```

A la hora de usar estos espacios para concretar el objeto de una clase, pondremos un espacio unido con un punto al nombre de la clase. Un ejemplo de ello podría ser:

```
MiEspacio.MiClase miobjeto = new MiEspacio.MiClase(parametros);
```

Para no tener que repetir continuamente el nombre del espacio de trabajo, que puede ser muy largo, el lenguaje tiene la directiva *using*, que permite definirlo una vez en la clase y utilizar después todo su contenido sin necesidad de volver a nombrarlo. Por ejemplo:

```
using MiEspacio;
...
MiClase miobjeto = new MiClase(parametros);
```

Para los espacios de nombres que tengan nombres largos también se pueden usar alias. De esta forma, cuando coincidan los nombres de clases en dos espacios de trabajo diferentes, al menos no habrá que repetir todo el nombre completo. Por ejemplo:

```
using ME1 = MiEspacio1;
using ME2 = MiEspacio2;
```

Clases

Todos los programas en *Visual C#* se organizan en clases, ya que éstas componen la base principal de cualquier aplicación.

Para construir una clase se utiliza la palabra clave *class*. Por ejemplo, una construcción de clase sería así:

```
class MiClase
{
    // Dentro va el contenido de la clase
}
```

En este ejemplo, *MiClase* es el nombre que adopta la clase creada y dentro de las llaves estaría el código para todos los miembros de la misma.

Las clases pueden ser públicas (lo que permite que se pueda acceder a ellas desde cualquier ensamblado) o privadas (sólo puede acceder a ellas el código que pertenece al mismo ensamblado). Las clases públicas llevarán la palabra clave *public* antes de *class* y las privadas llevarán la palabra clave *private*.

Variables y constantes

Las variables sirven para reservar un espacio de memoria para un objeto. Si dicho contenido tiene un valor fijo, entonces en lugar de una variable será una constante.

A la hora de declarar variables y constantes, siempre hay que indicar el tipo de datos que van a contener. La forma de declarar una variable sería similar a la del siguiente ejemplo:

```
string variable; // Se declara la variable de tipo string
```

También es posible inicializar la variable con un valor en la misma instrucción de la declaración. Por ejemplo:

```
string variable="Elena";
int numero=100;
```

3. Programación base

Por otra parte se tienen las constantes, que también reservan un espacio de memoria, pero que nunca van a poder variar su contenido. Para crear una constante, además del tipo de datos, hay que utilizar la palabra clave *const* que la define. Por ejemplo:

```
const double E=2.718281828459; // Constante número de Euler
```

Las variables y las constantes también pueden ser públicas (accesibles desde fuera del ámbito en el que están declaradas) o privadas (accesibles solamente en el ámbito de su declaración). Por defecto, si no se indica nada, todas ellas son privadas.

Una parte muy importante dentro de la programación es el buen uso de los tipos de datos. Cuanto mejor estén declarados, mejor será la eficiencia de uso de los recursos. Siempre hay que tratar de ajustar todo lo que se pueda el tamaño de almacenamiento de las variables y constantes en relación a los valores que vayan a contener. Hay que pensar en el valor máximo que se va a guardar y, en función de ello, elegir el tipo más adecuado.

Antes los tipos se solían dividir en tipos básicos o primitivos y tipos complejos, pero en este entorno de desarrollo esto ha cambiado. Se utiliza un sistema de tipos unificado llamado *CTS*, en el cual todos los tipos de datos son clases derivadas de la clase principal *System.Object*.

De todas formas, para que no todos los tipos sean tratados como objetos complejos y ello perjudique al rendimiento, se permite distinguir entre los tipos valor y los tipos referencia. Cuando se declara una variable de tipo valor, lo que se hace es reservar directamente un espacio de memoria que guarda el dato que contiene la variable. La única pega de usar el tipo valor es que no puede contener un valor nulo (algo que todos los objetos permiten), pero la eficacia es mayor y el rendimiento de la aplicación mejora.

En el caso de los tipos referencia, se reserva el espacio de memoria para el dato, pero lo que se devuelve es la referencia al objeto (un puntero al objeto). En este caso sí que puede contener un valor nulo, puesto que al ser una referencia a un lugar, el contenido sí que puede estar vacío.

Los tipos de datos que se pueden utilizar tanto en *Visual C#* como en los demás lenguajes que usan el mencionado *CTS* son los de la siguiente tabla:

Tipo CTS	Alias C#	Descripción	Contenido
<i>System.Object</i>	<i>object</i>	Objeto base	Cualquier objeto
<i>System.String</i>	<i>string</i>	Cadena de caracteres	Cualquier cadena
<i>System.SByte</i>	<i>sbyte</i>	<i>Byte</i> con signo	-128 a 127
<i>System.Byte</i>	<i>byte</i>	<i>Byte</i> sin signo	0 a 255
<i>System.Int16</i>	<i>short</i>	Entero de 2 <i>bytes</i> con signo	-32.768 a 32.767
<i>System.UInt16</i>	<i>ushort</i>	Entero de 2 <i>bytes</i> sin signo	0 a 65.535
<i>System.Int32</i>	<i>int</i>	Entero de 4 <i>bytes</i> con signo	-2.147.483.648 a 2.147.483.647
<i>System.UInt32</i>	<i>uint</i>	Entero de 4 <i>bytes</i> sin signo	0 a 4.294.967.295
<i>System.Int64</i>	<i>long</i>	Entero de 8 <i>bytes</i> con signo	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
<i>System.UInt64</i>	<i>ulong</i>	Entero de 8 <i>bytes</i> sin signo	0 a 18.446.744.073.709.551.615
<i>System.Char</i>	<i>char</i>	Caracteres Unicode de 2 <i>bytes</i>	0 a 65.535
<i>System.Single</i>	<i>float</i>	Valor de coma flotante de 4 <i>bytes</i>	1,5E-45 a 3,4E+38
<i>System.Double</i>	<i>double</i>	Valor de coma flotante de 8 <i>bytes</i>	5E-324 a 1,7E+308
<i>System.Boolean</i>	<i>bool</i>	Verdadero/falso	<i>true</i> o <i>false</i>
<i>System.Decimal</i>	<i>decimal</i>	Valor de coma flotante de 16 <i>bytes</i>	1E-28 a 7,9E+28
<i>System.DateTime</i>	<i>datetime</i>	Fecha y hora	Cualquier fecha y hora

La misma variable de un tipo básico puede declararse también como el tipo del objeto correspondiente. Por ejemplo, en cuanto a declaración de tipo, estas dos son equivalentes:

```
int numero=100;
System.Int32 numero=100;
```

3. Programación base

Otro tema importante cuando se habla de tipos de datos son las conversiones entre ellos. Hay veces que se quieren realizar operaciones entre tipos de datos diferentes, pero no siempre es posible directamente. Para ello se pueden convertir unos tipos de datos en otros. Por supuesto, no todas las conversiones son factibles.

Para los tipos básicos, basta con hacer un *cast* para que se realice la conversión. Se llama *cast* a poner el nombre del tipo delante del valor o de la variable entre paréntesis. Por ejemplo:

```
double numdoble = 123.456;  
int numentero = (double) 123.456;
```

En este caso, el valor que se guarda en la variable *numentero* es 123 y la parte decimal queda eliminada.

Para otras conversiones más complejas entre tipos se utilizan métodos propios de las clases de los tipos como objetos.

Operadores

Los operadores sirven para realizar cualquier tipo de operación con los argumentos que tengan y así obtener un nuevo resultado.

En la siguiente tabla se exponen los operadores existentes junto con la descripción de los mismos:

Operador	Descripción
(expresión)	Los paréntesis indican la precedencia de las operaciones.
objeto.elemento	El punto permite el acceso al elemento de un objeto. Dicho elemento puede ser una variable, un objeto o un método.
metodo(par1, par2, ..., parN)	Los paréntesis tras los métodos permiten especificar los parámetros que se aplican a dicho método, los cuales están separados por comas.
array[indice]	Los corchetes permiten pasar un índice a un <i>array</i> , de forma que se define uno de sus elementos.

Operador	Descripción
variable++	Con el doble + se incrementa en 1 el valor de la variable. Por ejemplo: <code>int a=10;</code> <code>int b=a++;</code> Tras la asignación, la variable <i>a</i> pasa a valer 11, aunque la variable <i>b</i> valdrá 10, puesto que el incremento es posterior a la asignación.
++variable	Igual que el anterior, pero el incremento se realiza antes de nada. Por ejemplo: <code>int a=10;</code> <code>int b=++a;</code> Tras la asignación, la variable <i>a</i> pasa a valer 11 y la variable <i>b</i> valdrá también 11, puesto que el incremento es anterior a la asignación.
variable--	Con el doble - se reduce en 1 el valor de la variable. Por ejemplo: <code>int a=10;</code> <code>int b=a--;</code> Tras la asignación, la variable <i>a</i> pasa a valer 9, aunque la variable <i>b</i> valdrá 10, puesto que el decremento es posterior a la asignación.
--variable	Igual que el anterior, pero el decremento se realiza antes de nada. Por ejemplo: <code>int a=10;</code> <code>int b=--a;</code> Tras la asignación, la variable <i>a</i> pasa a valer 9 y la variable <i>b</i> valdrá también 9, puesto que el incremento es anterior a la asignación.
new	El operador <i>new</i> permite crear nuevas instancias de objetos.
typeof	El operador <i>typeof</i> permite obtener el tipo de un objeto.
sizeof	El operador <i>sizeof</i> permite obtener el tamaño en <i>bytes</i> que ocupa un objeto.

3. Programación base

Operador	Descripción
checked	<p>El operador <i>checked</i> realiza un chequeo para comprobar si hay un desbordamiento a la hora de asignar un valor a una variable.</p> <p>Por ejemplo:</p> <pre>byte a=250; // El rango de valores es 0-255 checked {a=a+10;}</pre> <p>Tras la asignación, se produce un error de desbordamiento, puesto que el valor 260 obtenido no está dentro del rango y el chequeo provoca dicho error.</p>
unchecked	<p>El operador <i>unchecked</i> evita un chequeo para comprobar si hay un desbordamiento a la hora de asignar un valor a una variable.</p> <p>Por ejemplo:</p> <pre>byte a=250; // El rango de valores es 0-255 unchecked {a=a+10;}</pre> <p>Tras la asignación, no se produce un error de desbordamiento, puesto que se ha indicado que no hay que chequear, así que la variable <i>a</i> pasa a tener el valor 5 (260-255), que es el sobrante de lo que no entra en el rango de la variable.</p>
+variable	El operador + con una única variable simplemente sirve para ponerle el signo positivo a la variable.
-variable	El operador - con una única variable sirve para ponerle el signo negativo a la variable.
!	Este operador invierte el valor lógico de una variable booleana, pasando de cierto a falso y viceversa.
~	Este operador invierte el valor lógico de cada uno de los <i>bits</i> de una variable numérica, pasando de 0 a 1 y viceversa cada <i>bit</i> .
conversion (variable)	Este operador permite convertir una variable de un tipo a otro, siempre y cuando sea factible.
variable1+variable2	El operador + entre dos variables realiza la suma de ambas.
variable1-variable2	El operador - entre dos variables realiza la resta de ambas.
variable1*variable2	El operador * entre dos variables realiza la multiplicación de ambas.

Operador	Descripción
variable1/variable2	El operador / entre dos variables realiza la división entre ambas.
variable1%variable2	El operador % entre dos variables realiza la división entre ambas y devuelve el resto.
<<	Este operador realiza un movimiento de los <i>bits</i> , uno a uno, de una variable numérica hacia la izquierda. Por ejemplo: <code>byte a=25; // El valor es 00011001=25 a=a<<1; // El valor es 00110010=50</code>
>>	Este operador realiza un movimiento de los <i>bits</i> , uno a uno, de una variable numérica hacia la derecha. Por ejemplo: <code>byte a=25; // El valor es 00011001=25 a=a>>1; // El valor es 00001100=12</code>
<	Este operador relacional indica que una variable contiene un valor menor que otra.
>	Este operador relacional indica que una variable contiene un valor mayor que otra.
<=	Este operador relacional indica que una variable contiene un valor menor o igual que otra.
>=	Este operador relacional indica que una variable contiene un valor mayor o igual que otra.
=	Este operador relacional indica que dos variables contienen el mismo valor.
!=	Este operador relacional indica que dos variables contienen distinto valor.
is	Este operador permite saber si una variable es de un cierto tipo.
&&	Este operador realiza la operación <i>and</i> entre dos elementos booleanos y devuelve el resultado booleano. Sólo devolverá un valor cierto si ambos son ciertos.
	Este operador realiza la operación <i>or</i> entre dos elementos booleanos y devuelve el resultado booleano. Devolverá un valor cierto si alguno de ellos lo es.
&	Este operador realiza un <i>and</i> a nivel de <i>bits</i> , de forma que devuelve el resultado de una comparación como una nueva lista binaria de <i>bits</i> , poniendo un 1 cuando ambos valores del <i>bit</i> a sumar sea 1 y 0 en caso contrario.

3. Programación base

Operador	Descripción
	Este operador realiza un <i>or</i> a nivel de <i>bits</i> , de forma que devuelve el resultado de una comparación como una nueva lista binaria de <i>bits</i> , poniendo un 0 cuando ambos valores del <i>bit</i> a sumar sea 0 y 1 en caso contrario.
^	Este operador realiza un <i>xor</i> a nivel de <i>bits</i> , de forma que devuelve el resultado de una comparación como una nueva lista binaria de <i>bits</i> , poniendo un 1 cuando ambos valores del <i>bit</i> a sumar sean diferentes y 0 en caso contrario.
expresion?valorT: valorF	Este operador permite evaluar una expresión y en función de si es cierta o falsa, devuelve un resultado u otro.
=	Es el operador básico de asignación con el que se asigna un valor a una variable.
+=	Este operador de asignación realiza primero la operación de sumar a la variable el valor de la derecha de la igualdad y después asigna el resultado.
-=	Este operador de asignación realiza primero la operación de restar a la variable el valor de la derecha de la igualdad y después asigna el resultado.
*=	Este operador de asignación realiza primero la operación de multiplicar a la variable el valor de la derecha de la igualdad y después asigna el resultado.
/=	Este operador de asignación realiza primero la operación de dividir la variable por el valor de la derecha de la igualdad y después asigna el resultado.
%=	Este operador de asignación realiza primero la operación de dividir la variable por el valor de la derecha de la igualdad y después asigna el resto.
<<=	Este operador de asignación realiza primero la operación de mover los <i>bits</i> de la variable uno a uno a la izquierda tantas veces como indique el valor de la derecha de la igualdad y después asigna el resultado.
>>=	Este operador de asignación realiza primero la operación de mover los <i>bits</i> de la variable uno a uno a la derecha tantas veces como indique el valor de la derecha de la igualdad y después asigna el resultado.

Operador	Descripción
<code>&=</code>	Este operador de asignación realiza primero la operación <i>and bit a bit</i> entre la variable y el valor de la derecha de la igualdad y después asigna el resultado.
<code> =</code>	Este operador de asignación realiza primero la operación <i>or bit a bit</i> entre la variable y el valor de la derecha de la igualdad y después asigna el resultado.
<code>^=</code>	Este operador de asignación realiza primero la operación <i>xor bit a bit</i> entre la variable y el valor de la derecha de la igualdad y después asigna el resultado.

Métodos

Los métodos son los bloques de código que permiten manejar los datos de la clase en la que se han definido. La forma genérica de definir los métodos es la siguiente:

```
public Tipo MiMetodo(Tipo1 par1, Tipo2 par2, ...)  
{  
    // Dentro está el código del bloque de código del método  
}
```

Aunque por defecto está puesto *public* como accesibilidad para el método, también puede ser privado (*private*) y que su contenido sea únicamente interno a la clase. Eso sí, todos aquellos métodos de la clase que quieran ser utilizados desde otras clases, deberán definirse como públicos.

El tipo que aparece al principio es el tipo con el que se va a definir el dato que se devuelve como resultado. Es obligatorio que el método devuelva un dato del mismo tipo que está definido. Si el método que se está definiendo no va a devolver nada, entonces se utilizará la palabra clave *void* en lugar de poner un tipo de datos.

A continuación se pone el nombre del método, que es el que se va a utilizar para llamarlo. Por supuesto, no debe contener dicho nombre ni espacios ni caracteres extraños.

Lo siguiente son los parámetros del método. Cada uno de los parámetros debe ir precedido por su tipo de datos y después irá su nombre. De la

3. Programación base

misma forma que un método no tiene que devolver obligatoriamente un dato, tampoco tiene que llevar necesariamente parámetros de entrada. En este caso, se pondrían simplemente los paréntesis de abrir y cerrar.

Y lo que va después ya es el bloque de código correspondiente al método, que tendrá que estar encerrado entre las llaves de apertura y de cierre.

Otro aspecto interesante relativo a los métodos es la sobrecarga de los mismos. Eso quiere decir que se pueden definir métodos que tengan diferentes bloques de código en función de los parámetros que se les pase. Cuando se ejecuta la aplicación, se sabe que se quiere utilizar uno u otro bloque de código por la cantidad y el tipo de los parámetros del método.

No se considera sobrecarga cuando lo que es diferente es el tipo de datos del resultado. De hecho, es un problema y da error, porque el compilador, cuando realiza la llamada, no puede saber a cuál se está refiriendo en función de los parámetros si éstos son iguales.

A continuación se define un mismo método pero con sobrecarga, lo que nos da opción a llamarlo de tres maneras diferentes. En una de ellas varía el tipo de datos y en otra, la cantidad de parámetros.

```
public int Metodo1(int par1)
{
    // Bloque de código 1
}

public int Metodo1(int par1, int par2)
{
    // Bloque de código 2
}

public int Metodo1(String par1)
{
    // Bloque de código 3
}
```

A continuación se ponen tres ejemplos de llamadas para utilizar el mismo método modificando los parámetros:

```
int a=Metodo1(5);
int a=Metodo1("5");
int a=Metodo1(5, 5);
```

La primera de las llamadas tiene un único parámetro de tipo entero, con lo que el resultado se obtendrá de ejecutar el bloque de código 1.

La segunda llamada tiene un único parámetro, pero esta vez es de tipo cadena, porque va entrecomillado. En este caso el resultado se obtendrá de ejecutar el bloque de código 3.

Y la tercera llamada tiene dos parámetros de tipo entero, con lo que el resultado se obtendrá de ejecutar el bloque de código 2.

Otro tema que resulta interesante conocer a la hora de llamar a un método es la manera de introducir los parámetros, ya que existen dos posibilidades: por valor o por referencia.

Cuando se pasa un método por valor, lo que se hace es darle un dato que no va a variar fuera en ningún momento al finalizar el método, aunque dentro haya habido modificaciones.

Cuando se pasa un método por referencia, es como si se pasara un puntero a la zona de memoria en la que se encuentra el valor, con lo que si se modifica dicho valor, al finalizar la llamada sí que habrá habido modificación del dato.

Por supuesto, ello tiene sentido únicamente si lo que se pasa al método es una variable, no un dato fijo.

Para pasar un parámetro por referencia se utiliza la palabra clave *ref*. Un ejemplo de método con un parámetro de este tipo sería el de la siguiente declaración:

```
public int Doble(ref int par1)
{
    int res=0;
    par1=2*par1;
    res=par1;
    return res;
}
```

Vamos a ver ahora un ejemplo de llamada a este método desde otra parte del código, utilizando una variable externa.

```
int num=5;
int a=Doble(ref num);
```

3. Programación base

Cuando se ejecute la llamada, la variable *a* pasará a valer 10, puesto que dentro del método se ha doblado el valor del parámetro y se ha devuelto el doble del valor de entrada, que era 5. Pero lo interesante es lo que pasa con la variable *num*. Si el parámetro no estuviera definido por referencia, al terminar la llamada seguiría valiendo 5, porque no se modificaría nunca dicho valor. Sin embargo, en este caso se ha pasado por referencia y dentro del método se ha modificado el valor del parámetro, con lo que al salir de la llamada, la variable *num* también pasa a tener el valor 10.

Para pasar un valor por referencia es necesario que la variable que se introduzca esté previamente definida, ya que de lo contrario se provoca un error.

Puede darse el caso de que queramos utilizar una variable de salida que no se haya inicializado previamente. Para ello se utiliza la palabra clave *out* y funciona de la misma manera.

En el ejemplo anterior, se podría haber usado un nuevo parámetro que nos devolviera otro valor diferente. La declaración sería así:

```
public int DobleCuadruple(ref int par1, out int par2)
{
    int res=0;
    par1=2*par1;
    par2=2*par1;
    res=par1;
    return res;
}
```

La llamada a este nuevo método sería la siguiente:

```
int num=5;
int num4;
int a=DobleCuadruple(ref num, ref num4);
```

En este caso, los valores para las variables *a* y *num* serían los mismos de antes, 10. Pero la variable *num4* contendría tras la llamada el valor 20, puesto que dentro sí que se ha definido dicho resultado y tras la llamada se ha recibido como una salida más.

Por último, otra característica importante dentro de los métodos es si son estáticos o no. Los métodos estáticos son los que están declarados utilizando la palabra clave *static*. Este tipo de métodos son los que pueden ser ejecutados sin necesidad de instanciar la clase en la que están declarados.

El método *Main* o método principal de una clase tiene que ser obligatoriamente un método estático. Que tenga que ser así es muy lógico, porque para que en ejecución se pueda lanzar un método principal antes que nada, no se puede haber instanciado antes su clase.

Estos métodos estáticos no aparecen como miembros de las instancias de las clases, sino que integran directamente la propia clase.

Un ejemplo de declaración de un método estático sería el siguiente:

```
public static int Metodo1(int par1)
{
    // Bloque de código
}
```

Constructores

Un constructor de una clase es un método que ejecuta las primeras acciones de un objeto en el momento en que se crea la instancia de la clase. Este tipo de acciones suelen realizar tareas básicas como la inicialización de variables o el establecimiento de valores por defecto para las propiedades.

Los constructores tienen el mismo nombre que la clase y además no pueden devolver valores.

Lo primero que se ejecuta cuando se crea un objeto es el constructor de la clase a la que pertenece. El formato que deben tener es el siguiente:

```
namespace Constructores
{
    class MiObjeto
    {
        public MiObjeto()
        {
            // Instancia de la clase
        }
    }
}
```

--- 3. Programación base

```
class Aplicacion
{
    static void Main()
    {
        MiObjeto obj = new MiObjeto();
    }
}
```

En este caso, se tiene la clase *MiObjeto* y el constructor del mismo nombre, pero no ningún tipo que indique el resultado, porque no puede haberlo. En el método *Main* se ha creado el objeto y ahí se va a ejecutar la instancia.

Debido a que los constructores vienen a ser unos métodos especiales, también les está permitido tener ciertas cualidades de los métodos. Por ejemplo, se pueden sobrecargar. Se podría definir otro constructor que tuviera uno o más parámetros. En este caso, la propia construcción del objeto es la que define el constructor que se va a utilizar.

En el ejemplo anterior, se puede definir el constructor sobrecargado de la siguiente forma:

```
public MiObjeto(tipo parametro)
{
    // Instancia de la clase
}
```

Después se pueden crear objetos diferentes que ejecuten uno u otro método en función del parámetro de entrada. En las siguientes llamadas se puede ver un ejemplo:

```
MiObjeto obj1 = new MiObjeto();
MiObjeto obj2 = new MiObjeto(dato);
```

También se pueden definir constructores estáticos. Este tipo de constructores pueden realizar llamadas a métodos estáticos o inicializar variables estáticas.

Otro tema importante en este lenguaje que está relacionado con los constructores son los destructores. Cuando un objeto ya no se va a utilizar más, se puede destruir. Internamente existe un recolector de basura que va eliminando de la memoria los objetos que son destruidos. No lo hace al momento, sino que lo va liberando cuando le hace falta, pero lo importante es que al programador se le libera de esta tarea de gestionar la memoria como se hace en los lenguajes de bajo nivel.

Los objetos tienen un método *Finalize* que es el que realiza la destrucción de los mismos. Pero este método no se puede llamar desde el código ni tampoco se puede volver a escribir. Simplemente, el recolector de basura es el encargado de realizar las llamadas correspondientes a los métodos de destrucción, siendo transparente para el usuario.

Lo que se puede hacer es crear explícitamente un destructor, utilizando para ello la misma manera que los constructores, pero el nombre irá precedido por el símbolo ~.

En el ejemplo anterior, se declararía de la siguiente forma:

```
~MiObjeto()
{
    // Destructor de la clase
}
```

Dentro del código del destructor se pueden introducir instrucciones que se ejecutarán cuando el objeto se destruya. En el ejemplo anterior, cuando se termina la llamada al objeto, se acaba el programa. En ese momento pasará el recolector de basura y se encargará de ejecutar las acciones del destructor que se ha definido.

Lo mismo ocurre cuando en un programa más complejo se crea un objeto dentro de una función o de un método, siendo local a ellos. En el momento en que la función termine o el método acabe su ejecución, dicho objeto pasará a ser destruido, puesto que ya no se va a utilizar más si es local. Eso sí, hay que tener en cuenta, como se ha dicho antes, que el recolector

--- 3. Programación base ---

de basura no va a destruir los objetos inmediatamente, así que hay que tener mucho cuidado con el código que se define en los destructores, puesto que no se va a ejecutar de manera secuencial como el resto del código de la aplicación.

Otra posibilidad para liberar un objeto es asignarle el valor *null*. En ese momento la variable que contiene el objeto estará ya vacía, pero de la misma forma que antes, el destructor se ejecutará cuando el recolector de basura lo elimine, no en el momento de asignar el valor nulo.

También existe la posibilidad de ejecutar el recolector de basura cuando al desarrollador le interese. Para ello hay que realizar una llamada a *System.GC.Collect()*, que es el método recolector del objeto *Garbage Collector*. En ese momento sí que el compilador va a liberar de la memoria los objetos que antes hayan sido destruidos.

--- **Campos** ---

Los campos representan los datos de una clase y se construyen a base de indicadores. Los campos se deben declarar en la propia clase, pero no dentro de un método o función de la misma. Estos campos se deben declarar como públicos, de manera que sean accesibles desde fuera de la clase. Pueden tener valores variables o constantes.

Un ejemplo de una clase con varios campos puede ser el siguiente:

```
namespace Rectangulo
{
    class Rectangulo
    {
        public Rectangulo(double base, double altura)
        {
            this.Base=base;
            this.Altura=altura;
            this.Area=base*altura;
        }

        public double Base;
        public double Altura;
        public double Area;
    }

    class RectanguloApp
    {
        static void Main()
```

```
{  
    Rectangulo r;  
    double base=0;  
    double altura=0;  
  
    Console.WriteLine("Base del rectángulo: ");  
    base=Double.Parse(Console.ReadLine());  
  
    Console.WriteLine("Altura del rectángulo: ");  
    altura=Double.Parse(Console.ReadLine());  
  
    r=new Rectangulo(base, altura);  
  
    Console.WriteLine("El área es : " + r.Area);  
}  
}  
}
```

Dentro del método *Main* se realizan los accesos a los campos de la clase *Rectangulo*. La sintaxis para acceder a un campo de un objeto es de la forma *objeto.campo*, igual que cuando se llama a un método, pero sin usar los paréntesis finales.

Cuando el campo al que se quiere acceder es una constante, funciona como si fuera estático. No es necesario crear una instancia de un objeto y se puede usar directamente el nombre de la clase seguido de la constante.

En el ejemplo anterior, el cálculo del área se realiza en la propia clase, lo cual es lógico. Pero tal y como está definido el campo, dicho valor podría ser modificado *a posteriori*, es decir, el desarrollador podría asignar un valor directamente al campo *Area* y perdería todo el sentido.

Para solucionar este tipo de cosas, *Visual C#* permite crear variables que sean de sólo lectura. De esta manera, no se puede cambiar su valor ni por error. La declaración sería de la siguiente forma:

```
public readonly double Area;
```

Estos campos de sólo lectura solamente pueden ser asignados una vez en el constructor de la clase. Desde el método *Main* no se puede cambiar el valor de este tipo de campos.

Los campos, de la misma forma que los métodos y los constructores, también pueden ser estáticos. De la misma forma que en el resto, se utiliza esta característica cuando el campo tiene que ver más con la clase genérica que con un objeto que instancia a la clase. Eso sí, cuando un campo es estático,

Propiedades

Las propiedades son otra manera de representar los datos de los objetos de una clase. La principal ventaja respecto a los campos es que se tiene un control total de los valores que se reciben o se devuelven y no hay limitaciones a la hora de modificar sus contenidos el número de veces que el desarrollador quiera.

Internamente, las propiedades funcionan de la misma manera que los métodos (por supuesto, sin incluir los paréntesis en las llamadas), porque aunque se muestran como si fueran campos, ejecutan el bloque de código que se encuentra dentro de ellas.

Lo normal es no acceder a las propiedades directamente, sino crearlas como si fueran de acceso privado y después utilizar métodos para obtener y modificar su contenido.

La sintaxis de una propiedad estándar es la siguiente:

```
public tipo MiPropiedad
{
    get
    {
        // Se devuelve el contenido de la propiedad
        return Valor;
    }
    set
    {
        // Se asigna el nuevo valor a la propiedad
    }
}
```

Cada propiedad tiene que tener un tipo concreto y los datos que se utilicen para añadir contenido o para recibir un resultado deben ser del mismo tipo que el de la propiedad.

Dentro de la definición de la propiedad hay dos bloques de código: el bloque *get* y el bloque *set*. Dentro del bloque *get* se definen las acciones que puedan ser necesarias para obtener el valor de respuesta y después la propia devolución de la respuesta. Dentro del bloque *set* es donde se asigna el nuevo valor que va a recibir la propiedad. Ambos bloques de código deben estar siempre dentro de la propiedad.

Un caso particular serían las propiedades de sólo lectura. Para ello basta con que la propiedad no tenga el bloque de código *set*, con lo que únicamente podrá leerse su contenido, pero nunca actualizarse.

Control de flujo

En programación siempre son necesarias instrucciones de control de flujo que permitan que no todas las instrucciones se ejecuten una detrás de otra. Lo normal es que haya que realizar bifurcaciones en el código, de manera que en función de ciertas condiciones se pueda ejecutar una parte del código u otra.

La instrucción *if... else...*

La primera instrucción que cambia el control de flujo es la instrucción condicional, que básicamente usa las palabras clave *if* y *else*.

Para explicar su funcionamiento, lo más sencillo es ver un ejemplo básico de la vida cotidiana.

```
si (sacas buenas notas)
{
    nos vamos todos de viaje
}
si no
{
    te quedarás estudiando en vacaciones
}
```

Suponiendo que lo anterior fuera parte de un programa, lo que está claro es que dependiendo de si la condición se evalúa como cierta o como falsa, se ejecutará una parte del código o la otra. Con la condición evaluada a cierto se ejecutará la parte *si* y con la condición evaluada a falso se ejecutará la parte *si no*.

3. Programación base

Vamos a ver un ejemplo similar, pero utilizando código en *Visual C#*, de manera que haya un trozo de código que dependa de la evaluación de una condición.

```
if (num>=10)
{
    Console.WriteLine(«El número tiene varios dígitos»);
}
else
{
    Console.WriteLine(«El número tiene un dígito»);
}
```

Suponiendo que la variable *num* fuera una variable numérica y que contuviera un valor entero positivo, el programa cambiaría el control de flujo en función de si dicha variable contiene un valor inferior a 10 o superior o igual a 10. Si es 10 o más, entonces se va a escribir que tiene varios dígitos. Si el número es inferior a 10, entonces se va a escribir que tiene un único dígito.

Hay que tener en cuenta que cuando el bloque de código no contiene más que una instrucción, no es necesario poner las llaves. En el caso anterior, funcionaría igualmente de la siguiente forma:

```
if (num>=10)
    Console.WriteLine(«El número tiene varios dígitos»);
else
    Console.WriteLine(«El número tiene un dígito»);
```

De todas formas, es recomendable utilizar las llaves por norma. Es muy común que después se quiera poner un comentario o cualquier cosa delante de una de las líneas y nos olvidemos de poner las llaves, así que como vale más prevenir, será mejor que estén siempre aunque no sean imprescindibles.

Estas instrucciones condicionales también pueden ir encadenadas. Si no se cumple la primera condición, puede que la segunda parte del código también quiera llevar otra nueva condición e incluir dos nuevos bloques de código que sea posible ejecutar. Con la encadenación de instrucciones *if* se soluciona el problema

Un ejemplo de instrucciones encadenadas es el siguiente:

```
if (num>=100)
{
    Console.WriteLine("El número tiene más de dos dígitos");
}
else
{
    if (num>=10)
    {
        Console.WriteLine("El número tiene dos dígitos");
    }
    else
    {
        Console.WriteLine("El número tiene un dígito");
    }
}
```

En este caso se ha incluido la segunda instrucción condicional dentro del segundo bloque de código, pero también es posible combinar directamente la instrucción condicional con la palabra clave *else*, obteniendo un *else if*. El mismo ejemplo quedaría así:

```
if (num>=100)
{
    Console.WriteLine("El número tiene más de dos dígitos");
}
else if (num>=10)
{
    Console.WriteLine("El número tiene dos dígitos");
}
else
{
    Console.WriteLine("El número tiene un dígito");
}
```

3. Programación base

Si se cumple la primera condición, entonces ya no se evaluará la segunda condición, puesto que ya se sabe que no va a entrar por ninguno de los otros dos bloques de código. Pero si no se cumple la primera condición, entonces deberá evaluarse la segunda para que se ejecute uno de los dos bloques de código restantes.

Por supuesto, las expresiones que se utilicen para evaluar si se entra por uno u otro bloque de código pueden ser muy complejas y contener varias condiciones. Utilizando los operadores lógicos para unir condiciones, se pueden poner todas las que se necesiten.

Existe también un tipo de instrucción *if* un tanto especial y específica. Se utiliza cuando se quiere realizar una asignación a una variable en función de una condición. En este caso, se puede escribir la misma instrucción, pero de manera simplificada.

Este tipo de instrucción usando el *if* convencional se escribe así:

```
if (condicion)
{
    variable=valor1;
}
else
{
    variable=valor2;
}
```

Otra manera de escribir esto mismo usando el formato abreviado es:

```
variable=(condicion) ? valor1: valor2;
```

Es decir, si la condición se evalúa como cierta, se asigna el valor que aparece después del símbolo ?, pero si se evalúa como falsa, entonces se asigna el valor que aparece después del símbolo :.

La instrucción *switch*

Otra instrucción que sirve para el control de flujo es la instrucción *switch*. Esta instrucción funciona más o menos como una serie de instrucciones condicionales anidadas, pero con la restricción de que la expresión siempre

se evalúa con la misma condición. Lo que varía en cada caso es el resultado que puede tener la condición, pero siempre sobre la misma variable, es decir, que no se pueden poner condiciones independientes para cada uno de los bloques de código que después se van a ejecutar.

Para ver su funcionamiento, también en este caso se va a ver un ejemplo básico de la vida cotidiana.

```
comprobar (sobresalientes que sacas en las notas)
{
    en caso de ser 1:
        tienes un premio pequeño
    en caso de ser 2:
        tienes un premio mediano
    en caso de ser 3:
        tienes un premio grande
    en otro caso:
        haremos un viaje fantástico
}
```

Suponiendo que lo anterior fuera parte de un programa, se aprecia que, dependiendo de si el número de sobresalientes es uno u otro, se ejecutará una parte del código u otra. Con la condición evaluada a cierto se ejecutará la parte que corresponda con ese valor y, si no se cumple ninguno de los casos, entonces se ejecutará la parte del final.

Una cosa que hay que tener en cuenta es que al pasar a código de *Visual C#*, cuando se termina de ejecutar un bloque de instrucciones, hay que indicarle específicamente que no se quiere hacer nada más, porque de lo contrario, se seguirán ejecutando las instrucciones de la siguiente condición.

Lo mejor es verlo con un ejemplo de código real. Una transformación más o menos similar del pseudocódigo anterior sería la siguiente:

```
switch (numSobresalientes)
{
    case 1:
        premio=10;
        break;
    case 2:
```

3. Programación base

```
    premio=25;
    break;
case 3:
    premio=50;
    break;
default:
    premio=1000;
}
```

Como se puede apreciar, después de cada uno de los bloques de código se ha incluido la instrucción *break*. Esta instrucción hace que el punto de ejecución del código se vaya al final de la instrucción *switch*.

En esta instrucción no es necesario utilizar las llaves de apertura y cierre para establecer los bloques de código, puesto que se asume que el bloque ocupa las líneas que hagan falta hasta que aparezca un *break* o hasta que venga la siguiente condición.

Si lo que se quiere es que dos condiciones diferentes ejecuten el mismo bloque de código, entonces se pueden poner una detrás de la otra, sin que haya ni instrucciones ni *break* en la primera de ellas. Por ejemplo, así:

```
switch (numSobresalientes)
{
    case 1:
    case 2:
        premio=25;
        break;
    case 3:
        premio=50;
        break;
    default:
        premio=1000;
}
```

En el ejemplo anterior, tanto si el número de sobresalientes es 1 como si es 2, el el valor del premio será 25.

Otra restricción que tiene la sentencia *switch* es que el tipo de la variable que se va a evaluar no puede ser uno cualquiera. Los tipos que están permitidos son:

- *sbyte*
- *byte*
- *short*
- *ushort*
- *int*
- *uint*
- *long*
- *ulong*
- *char*
- *string*

Si se tiene un tipo diferente, pero que se puede convertir a uno de los anteriores, entonces también es una opción válida realizar la conversión.

Estructuras iterativas

Otro tipo de instrucciones imprescindibles para desarrollar aplicaciones son las estructuras iterativas. Este tipo de instrucciones permiten repetir la ejecución de un bloque de código hasta que se cumpla una determinada condición o un número concreto de veces. Normalmente a estas instrucciones se las llama *bucles*.

El bucle *for*

Lo que hacen este tipo de bucles es ir asignando valores a una variable que actúa de contador y avanza desde un valor inicial hasta un valor final. Cuando la variable obtiene un valor que está fuera del intervalo, el bucle termina.

3. Programación base

Normalmente sirve para ejecutar un número determinado de veces el bloque de código que lo contiene.

La sintaxis de este tipo de bucles es la siguiente:

```
for (variable=valorInicial; condicion; valorSiguiente)
{
    // Se ejecuta el bloque de código con el valor actual
}
```

Como se puede apreciar en la instrucción, se empieza por la palabra clave *for* y después entre paréntesis se indican tres partes diferenciadas. El separador para dichas partes es el punto y coma. En la primera de ellas, a la variable del bucle se le asigna el valor con el que se va a ejecutar la primera vuelta del bucle. En la segunda parte se coloca una condición que permite salir del bucle en cuanto deja de cumplirse. Por último, en la tercera parte se modifica el valor de la variable. Esta modificación se realiza cada vez que el bloque de código ha terminado de ejecutarse y vuelve a empezar el bucle. Cada vez que se realiza la modificación, se vuelve a verificar la condición de la segunda parte para saber si se debe volver a ejecutar el bloque de código o se debe salir del bucle.

Por ejemplo, se quiere realizar la suma de los 10 primeros números enteros y guardarla en una variable. Se realizaría de la siguiente forma:

```
int suma=0;
for (byte iCont=1; iCont<=10 ; iCont++)
{
    suma=suma+iCont; // suma+=iCont;
}
```

Cuando se ejecute este código, la variable contadora *iCont* empezará con el valor 1 y se irá incrementando su valor en cada vuelta hasta llegar a 10. Cuando obtenga el valor 11 se terminará la iteración. En cada vuelta se sumará el valor en la variable *suma*, con lo que el resultado final que contendrá la variable será el resultado de la siguiente operación:

$$suma = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;$$

El bucle se ha hecho con valores de menor a mayor, pero también se podría haber hecho al contrario. Se puede empezar desde el valor superior e irlo reduciendo. El mismo ejemplo anterior sería así:

```
int suma=0;
for (byte iCont=10; iCont>=1 ; iCont--)
{
    suma=suma+iCont; // suma+=iCont;
}
```

En este caso, la instrucción es la misma, pero los valores van variando al contrario. El resultado final debe ser el mismo, aunque la operación que se realiza en este caso es la siguiente:

suma=10+9+8+7+6+5+4+3+2+1;

De la misma forma que en las instrucciones de control de flujo, las estructuras iterativas también pueden estar anidadas.

Por ejemplo, si se quiere que se dibujen los resultados de las tablas de multiplicar del 1 al 10, se podría realizar un bucle dentro de otro que tenga dos variables que vayan tomando valores del 1 al 10 y lo multipliquen. Podría ser como el siguiente ejemplo:

```
for (byte iCont=1; iCont<=10 ; iCont++)
{
    Console.WriteLine("Tabla del " + iCont);
    for (byte iCont2=1; iCont2<=10 ; iCont2++)
    {
        Console.Write((iCont*iCont2) + "   ");
    }
    Console.WriteLine("");
}
```

Analizando un poco el funcionamiento de estos dos bucles anidados se puede deducir cómo va a ser la salida por pantalla. Como se ve, en la parte principal es donde se realiza la operación de la multiplicación. Se escribe el número del resultado y después unos espacios en blanco para

--- 3. Programación base

que no queden todos los números pegados. Durante las 10 vueltas del bucle interior, se va escribiendo en la misma línea. Antes del bucle interior se realiza una escritura que indica la tabla del número que se va a escribir. Después del bucle interior se realiza un salto de línea, para que después de la lista de resultados de cada tabla se comience en una nueva línea de salida.

El bloque de código interior se va a ejecutar 100 veces (10 veces del bucle interior multiplicadas por 10 veces del bucle exterior).

El resultado por pantalla va a ser como el siguiente:

```
Tabla del 1
1   2   3   4   5   6   7   8   9   10
Tabla del 2
2   4   6   8   10  12  14  16  18  20
Tabla del 3
3   6   9   12  15  18  21  24  27  30
...
Tabla del 10
10  20  30  40  50  60  70  80  90  100
```

--- **El bucle while**

Este bucle está pensado más bien para que el bloque de código se ejecute hasta que se cumpla una determinada condición, pero no se suele usar para realizar un número fijo de vueltas. Aunque cualquiera de las estructuras iterativas puede servir para cualquier cosa.

La sintaxis de la instrucción *while* es la siguiente:

```
while (condicion)
{
    // Código a ejecutar mientras se cumpla la condición
}
```

Como se ha indicado, se puede realizar el mismo bucle que se ha hecho en el punto anterior usando *for*, pero utilizando *while*. A continuación, un

ejemplo de cómo puede hacerse el mismo bucle para sumar los 10 primeros números.

```
int suma=0;
byte iCont=1;
while (iCont<=10)
{
    suma=suma+iCont; // suma+=iCont;
    iCont++;
}
```

En este tipo de bucles, si la condición no se cumple la primera vez que se comprueba, el bloque de código no se llegará a ejecutar nunca.

El bucle *do*

Este bucle es prácticamente igual al anterior, con la diferencia de que tiene la condición al final del bucle. Ello implica que, como mínimo, se va a ejecutar una vez el bloque de código.

La sintaxis de la instrucción *do* es la siguiente:

```
do
{
    // Código a ejecutar mientras se cumpla la condición
} while (condicion);
```

El equivalente al ejemplo de los puntos anteriores sería el siguiente:

```
int suma=0;
byte iCont=1;
do
{
    suma=suma+iCont; // suma+=iCont;
    iCont++;
} while (iCont<=10);
```

3. Programación base

Como se puede apreciar, es prácticamente igual cuando se va a entrar al menos una vez en el bloque de código. En este ejemplo se sabe que como mínimo se va a realizar una suma, pero como en este caso la iteración se va a producir 10 veces, no cambia nada con respecto al ejemplo del bucle *while*.

Rupturas de código

La ejecución del código no tiene que ser siempre secuencial y para realizar un salto a otra ubicación se pueden usar las instrucciones de salto.

Ya en la instrucción *switch* se ha utilizado la instrucción *break*, que permite saltar al final de la instrucción. De manera similar se puede utilizar dentro de los bucles. Si dentro del bloque de código de un bucle aparece una instrucción *break*, lo que hace es salir directamente a la instrucción posterior al bucle. Ni siquiera se evalúa de nuevo la condición de salida.

Siguiendo con el mismo ejemplo de los puntos anteriores, otra posibilidad para realizar la suma de los 10 primeros números sería la siguiente:

```
int suma=0;
byte iCont=1;
while (true)
{
    suma=suma+iCont; // suma+=iCont;
    if (iCont>=10)
    {
        break;
    }
    iCont++;
}
```

En este caso se ha utilizado una condición que directamente se evalúa como cierta siempre. Este bucle sería infinito, salvo que en algún momento se produjera una ruptura. Se van sumando los números, pero cuando se llega a la condición que se está evaluando dentro del propio bucle, se llama a la instrucción de ruptura para que salga del bucle.

Otra instrucción que permite realizar un salto dentro de un bucle es la instrucción *continue*. A diferencia de la anterior, esta instrucción no salta directamente al punto posterior al bucle, sino que salta a la expresión

que se debe evaluar de nuevo. Lo que hace es evitar que se ejecuten las instrucciones posteriores hasta el final del bloque de código del bucle.

Por ejemplo, si se quiere realizar la suma de los números del 1 al 10, pero solamente los impares, se podría utilizar el bucle de la siguiente forma:

```
int suma=0;
byte iCont=0;
while (iCont<10)
{
    iCont++;
    if (iCont%2==0)
    {
        continue;
    }
    suma=suma+iCont; // suma+=iCont;
}
```

En este caso, cada vez que se entra en el bucle se comprueba si el número es par (el resto de la división por 2 del número es 0). Si se detecta que es par, no se quiere realizar la suma, así que se indica que continúe con la siguiente evaluación del bucle.

Hay que tener cuidado con varias cosas que se pueden apreciar en este ejemplo. Esta vez la variable de los sumandos se ha inicializado con el valor 0 y nada más entrar en el bucle, se ha incrementado para obtener el valor 1. Esto es así porque es imprescindible que el incremento del sumando se realice antes de chequear si debe continuar. ¿Qué pasaría si se hubiera hecho como en el ejemplo anterior?

```
int suma=0;
byte iCont=1;
while (iCont<10)
{
    if (iCont%2==0)
    {
        continue;
    }
    suma=suma+iCont; // suma+=iCont;
    iCont++;
}
```

--- 3. Programación base

La primera vez se suma el valor 1 y se incrementa la variable del sumando a 2, pero en la segunda vuelta, se evalúa la condición de par a cierta y se obliga a continuar de nuevo con la expresión. Como en este caso no se han ejecutado las instrucciones posteriores (las que están después del *if* en el ejemplo), la variable del sumando siempre va a valer 2 y siempre se va a volver arriba, con lo que el bucle sería infinito y no terminaría nunca.

Existe otra instrucción que permite saltar a otra parte del código y es la instrucción *goto*. Visual C# permite poner etiquetas en el código y usar esta instrucción para saltar a ellas. Esta instrucción no es nada recomendable dentro de la programación estructurada y muchos programadores están totalmente en contra de utilizarla. Por ello, simplemente se debe saber que existe, pero si se quiere utilizar, debe hacerse con muchísimo cuidado para que el código no pase a ser ilegible.

Recursividad

La recursividad es un recurso que tienen las funciones dentro de la programación y que les permite llamarse a sí mismas, normalmente modificando sus parámetros de llamada. De alguna manera se podría realizar una comparativa con los bucles y los saltos dentro de ellos.

Lo mejor es verlo con un ejemplo. Se puede hacer el mismo ejemplo de la suma de los 10 primeros números de la siguiente forma:

```
static int Sumar(byte iCont)
{
    if (iCont==1)
    {
        return 1;
    }
    return iCont+Sumar(iCont-1); // Se suma el siguiente número
}
```

Si se quieren sumar los 10 primeros números, habría que llamar a la función pasándole como parámetro dicho número. Sería así:

```
int suma=Sumar(10);
```

Lo que hace la función es sumar desde 10 hasta 1, que es el valor en el que se ha puesto el punto de parada. Cuando se llama a la función, empieza comprobando si se ha llegado al valor 1. Como no es el caso, lo

que se hace es sumar 10 al resultado de volver a llamar a la función, pero con el valor 9. A su vez, esta segunda llamada producirá que se sume el 9 a la propia función con el valor 8. Y así sucesivamente, hasta llegar al valor 1, que es cuando directamente se devuelve un 1 y se terminan las llamadas recursivas.

El resultado de la ejecución paso a paso sería el siguiente:

```
Sumar(10)=10+Sumar(9);  
Sumar(10)=10+9+Sumar(8);  
Sumar(10)=10+9+8+Sumar(7);  
Sumar(10)=10+9+8+7+Sumar(6);  
Sumar(10)=10+9+8+7+6+Sumar(5);  
Sumar(10)=10+9+8+7+6+5+Sumar(4);  
Sumar(10)=10+9+8+7+6+5+4+Sumar(3);  
Sumar(10)=10+9+8+7+6+5+4+3+Sumar(2);  
Sumar(10)=10+9+8+7+6+5+4+3+2+Sumar(1);  
Sumar(10)= 10+9+8+7+6+5+4+3+2+1;
```

Arrays

Hasta ahora se han visto las variables de programación que contienen dentro un dato únicamente. Los *arrays* permiten almacenar varios datos en la misma variable. Los datos que se almacenan deben ser del mismo tipo.

Los datos que se incluyen en un *array* estarán guardados en diferentes posiciones. Para acceder a un valor concreto, es necesario utilizar su posición. A esa posición se le llama índice.

Los valores de los índices empiezan desde cero, es decir, que para acceder al primer elemento del *array* se usa el índice 0, para acceder al segundo elemento se usa el índice 1 y así sucesivamente. En general, para acceder al elemento *i* del *array* se usa el índice *i-1*.

Para declarar una variable de tipo *array* se usa la siguiente sintaxis:

```
tipo[] miTabla;
```

Como se puede apreciar, la diferencia entre declarar una variable normal y declarar un *array* es que después del tipo de datos se usan corchetes de apertura y cierre.

3. Programación base

A parte de la declaración de la variable, antes de poder usarla es necesario instanciar dicha variable. Ello supone que hay que indicar el número de elementos que se quieren guardar en el *array*. Lo mejor es ver un ejemplo de declaración e instanciación:

```
string[] misElementos;  
// Declaración del array  
misElementos = new string[5];  
// Instanciación del array
```

En este ejemplo se ha declarado la variable *misElementos* y después se ha indicado que la cantidad de elementos que se van a tener guardados en la variable van a ser 5. Como se ha indicado antes, el primer elemento será *misElementos[0]* y el quinto y último elemento será *misElementos[4]*.

A la hora de instanciar, el número de elementos también puede provenir de una variable. En función de las necesidades, se puede modificar el valor de la cantidad de elementos y así crear la instancia con el número de elementos deseado. El ejemplo anterior se podría haber declarado de la siguiente manera:

```
int numElementos = 5;  
// Cantidad de elementos del array  
string[] misElementos;  
// Declaración del array  
misElementos = new string[numElementos];  
// Instanciación
```

Por supuesto, también se pueden instanciar directamente en el mismo momento de declarar la variable.

```
int numElementos = 5;  
// Cantidad de elementos del array  
string[] misElementos = new string[numElementos];
```

Para asignar los valores a un *array*, éstos se deben incluir separados por comas y entre llaves de apertura y cierre. Se puede ver en este ejemplo:

```
int numElementos = 5;  
// Cantidad de elementos del array  
string[] misElementos = {"Elemento1", "Elemento2", "Elemento3",  
"Elemento4", "Elemento5"};
```

En este ejemplo no es necesario realizar una instancia para indicar el número de elementos, ya que al haberle asignado directamente un número de elementos concreto, ya se sabe la cantidad como si se hubiera instanciado.

Si se quiere obtener el contenido de un elemento concreto del array, basta con mirar el índice correspondiente. Por ejemplo:

misElementos[2] ↗ "Elemento3"

Un método importante de este tipo de variables es el método *length*, que devuelve la cantidad de elementos que tiene el *array*. En el caso anterior, *misElementos.length* devuelve el valor 5.

Por supuesto, si se realiza una nueva instancia de un *array*, el contenido que tenga hasta ese momento desaparece.

Los *arrays* no tienen que tener una única dimensión. Si se declaran varias dimensiones, se crean los *arrays* multidimensionales. La sintaxis de creación para un *array* de dos dimensiones es la siguiente:

tipo[,] miTablaMultidimensional;

Cada dimensión puede tener un número de elementos diferente. Un ejemplo de la creación de un *array* es el siguiente:

```
int indice1 = 5;  
// Cantidad de elementos de la primera dimensión  
int indice2 = 10;  
// Cantidad de elementos de la segunda dimensión  
string[,] misElementos = new string[indice1, indice2];
```

Por ejemplo, se puede querer guardar en un *array* los elementos de las tablas de multiplicar del 1 al 10. El contenido sería el siguiente:

3. Programación base

Índice	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	10
1	2	4	6	8	10	12	14	16	18	20
2	3	6	9	12	15	18	21	24	27	30
3	4	8	12	16	20	24	28	32	36	40
4	5	10	15	20	25	30	35	40	45	50
5	6	12	18	24	30	36	42	48	54	60
6	7	14	21	28	35	42	49	56	63	70
7	8	16	24	32	40	48	56	64	72	80
8	9	18	27	36	45	54	63	72	81	90
9	10	20	30	40	50	60	70	80	90	100

Para llenar un *array* de este tipo, se pueden utilizar los bucles anidados vistos en este mismo capítulo. Lo que sí hay que tener muy en cuenta es que los índices de los *arrays* empiezan desde cero y tienen un valor inferior en una unidad al valor del propio índice. Lo mejor es ver las asignaciones con un ejemplo.

```
string[,] tablasMultiplicar = new string[10, 10];
for (byte iCont=1; iCont<=10 ; iCont++)
{
    for (byte iCont2=1; iCont2<=10 ; iCont2++)
    {
        tablasMultiplicar[iCont-1, iCont2-1] = iCont*iCont2;
    }
}
```

Como se aprecia en el código, los bucles se realizan con los valores del 1 al 10, pero a la hora de realizar las asignaciones, se resta 1 a cada índice, para que puedan ir del 0 al 9. Al finalizar las asignaciones, se puede obtener el valor de un elemento cualquiera. Por ejemplo:

tablasMultiplicar[7, 4] → 40

Cada *array* puede ser de un tipo cualquiera de los existentes, lo que permite que también el tipo sea a su vez otro *array*. Es decir, que se pueden realizar *arrays* de *arrays*. En este caso, cada elemento de *array* contiene a su vez otro *array* completo.

Aunque este tipo de variables pueden ser muy similares a los *arrays* multidimensionales, pueden tener sus ventajas en casos concretos. Utilizando los bucles anidados, se pueden recorrer los elementos de manera bastante sencilla.

Una ventaja interesante es que se puede declarar el *array* principal como de tipo *object* y después declarar los *arrays* internos de tipos diferentes, ya que dichos tipos siempre van a ser compatibles con el tipo genérico de objeto.

Conversiones de tipos

En muchas ocasiones es necesario realizar conversiones de unos tipos a otros. Hay que tener en cuenta que en función de los tipos, los resultados de algunas operaciones pueden variar.

Los tipos numéricos dan mucho juego, puesto que es muy diferente, por ejemplo, dividir dos números enteros o dos números con decimales. Para no obtener resultados indeseados, hay que tener mucho cuidado con la definición de los tipos de variables numéricas y utilizar adecuadamente las operaciones aritméticas que proporciona el lenguaje.

El lenguaje *Visual C#* proporciona dos palabras clave, *implicit* y *explicit*, que especifican si se quiere realizar una conversión de tipos implícita o explícita. No todos los tipos son compatibles con los demás tipos, así que siempre hay que tener en cuenta que para realizar conversiones implícitas, los tipos deben ser compatibles entre ellos.

A continuación se incluye una tabla con las conversiones numéricas implícitas predefinidas.

Tipo	Tipos compatibles
<i>sbyte</i>	<i>short, int, long, float, double, decimal</i>
<i>byte</i>	<i>short, ushort, int, uint, long, ulong, float, double, decimal</i>
<i>short</i>	<i>int, long, float, double, decimal</i>

3. Programación base

<i>ushort</i>	<i>int, uint, long, ulong, float, double, decimal</i>
<i>int</i>	<i>long, float, double, decimal</i>
<i>uint</i>	<i>long, ulong, float, double, decimal</i>
<i>long</i>	<i>float, double, decimal</i>
<i>ulong</i>	<i>float, double, decimal</i>
<i>char</i>	<i>ushort, int, uint, long, ulong, float, double, decimal</i>
<i>float</i>	<i>double</i>

Se debe tener en cuenta que en algunos casos la conversión va a ca a ocasionar precisión en el resultado.

Como se puede apreciar en la tabla, no existen conversiones implícitas hacia el tipo *char*. Tampoco hay conversiones implícitas entre los tipos de punto flotante y el tipo *decimal*.

Y la siguiente es la tabla con las conversiones explícitas, que permiten convertir cualquier tipo numérico en otro tipo numérico diferente para el que no haya una conversión implícita.

Tipo	Tipo de conversión explícita
<i>sbyte</i>	<i>byte, ushort, uint, ulong, char</i>
<i>byte</i>	<i>sbyte, char</i>
<i>short</i>	<i>sbyte, byte, ushort, uint, ulong, char</i>
<i>ushort</i>	<i>sbyte, byte, short, char</i>
<i>int</i>	<i>sbyte, byte, short, ushort, uint, ulong, char</i>
<i>uint</i>	<i>sbyte, byte, short, ushort, int, char</i>
<i>long</i>	<i>sbyte, byte, short, ushort, int, uint, ulong, char</i>
<i>ulong</i>	<i>sbyte, byte, short, ushort, int, uint, long, char</i>
<i>char</i>	<i>sbyte, byte, short</i>
<i>float</i>	<i>sbyte, byte, short, ushort, int, uint, long, ulong, char, decimal</i>
<i>double</i>	<i>sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal</i>
<i>decimal</i>	<i>sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double</i>

Hay que tener en cuenta que las conversiones numéricas explícitas pueden producir una pérdida de precisión e incluso provocar excepciones en el programa.

Cuando se convierte un valor *decimal* a un valor entero, el valor original se redondea al valor entero más próximo. Puede ser hacia arriba o hacia abajo. Sin embargo, cuando se tiene un valor *double* o *float* y se convierte a un tipo entero, lo que se hace es truncar el valor, eliminando directamente todos los decimales que tenga.

Si la conversión de un valor *double* se hace a *float*, entonces se redondea al valor más próximo.

Si lo que se convierte es un valor *float* o *double* a *decimal*, entonces se convierte en una representación *decimal* redondeando al número más próximo tras la vigésimo octava posición decimal en caso de ser necesario.

Si se convierte un valor *decimal* en *float* o *double*, se redondea al valor más próximo del tipo destino.

Interfaces

Una interfaz es un conjunto de miembros que van a ser comunes entre diferentes clases. La interfaz debe saber cómo es la funcionalidad de las clases implementadas, aunque desconoce los detalles de la implementación, ya que no se puede implementar código en una interfaz.

Las interfaces describen un conjunto de miembros. Una ventaja que tienen es que en la misma clase se pueden implementar varias interfaces.

Práctica paso a paso

Queremos hacer primero una programación en pseudocódigo que, dado un valor del 0 al 10 contenido en una variable numérica, obtenga la nota escolar correspondiente en función de la siguiente lista:

- 0, 1, 2 → muy deficiente
- 3, 4 → insuficiente
- 5, 6 → aprobado

3. Programación base

- 7, 8 → notable
- 9, 10 → sobresaliente

Debemos guardar la nota en una variable y después escribirla por pantalla. Una vez que tengamos el pseudocódigo, tendremos que convertirlo todo a código de *Visual C#*.

De entrada, vamos a declarar una variable que es la que va a contener la nota numérica correspondiente:

entero nota

Declararemos también la variable que va a contener el resultado:

cadena resultado

Después vamos a utilizar una instrucción condicional de comprobación que permita usar los diferentes valores:

```
comprobar (valor de la nota)
{
    en caso de ser 0:
        en caso de ser 1:
            en caso de ser 2:
                guardar un muy deficiente en el resultado
                terminar comprobación
            en caso de ser 3:
            en caso de ser 4:
                guardar un insuficiente en el resultado
                terminar comprobación
            en caso de ser 5:
            en caso de ser 6:
                guardar un aprobado en el resultado
                terminar comprobación
            en caso de ser 7:
            en caso de ser 8:
                guardar un notable en el resultado
                terminar comprobación
            en caso de ser 9:
            en caso de ser 10:
                guardar un sobresaliente en el resultado
                terminar comprobación
            en otro caso:
                la nota no es válida
```

Por último, vamos a escribir el resultado:

Escribir resultado por pantalla

A continuación, traducimos todo el código a *Visual C#* para tenerlo implementado.

```
int nota=7;           // Se debe inicializar con el valor deseado
String resultado;
switch (nota)
{
    case 0:
    case 1:
    case 2:
        resultado="Muy deficiente";
        break;
    case 3:
    case 4:
        resultado="Insuficiente";
        break;
    case 5:
    case 6:
        resultado="Aprobado";
        break;
    case 7:
    case 8:
        resultado="Notable";
        break;
    case 9:
    case 10:
        resultado="Sobresaliente";
        break;
    default:
        resultado="La nota no es válida";
}
Console.WriteLine("Nota: " + resultado);
```

Ejercicios

Ejercicio 3.1

Escribe un código en *Visual C#* que empiece creando dos matrices bidimensionales con los siguientes contenidos:

5	6	4	2	8
1	9	7	3	5
2	9	0	5	7
1	6	8	4	9
3	6	7	4	5
3	0	5	9	4
8	1	2	6	7
5	9	4	8	3
1	2	0	6	5
8	9	4	2	7

Después escribe el código que realice la suma de las dos matrices y guarde el resultado en una nueva matriz unidimensional.

Ejercicio 3.2

Utilizando las mismas matrices del ejercicio anterior, realiza la resta de ellas, pero utilizando necesariamente bucles *while*.

Programación gráfica

CAPÍTULO
4

Programación gráfica

Programación orientada a eventos

Hasta ahora se ha hablado de la programación orientada a objetos, pero dentro de la programación gráfica, es muy importante también el concepto de programación orientada a eventos.

Los lenguajes visuales son lenguajes orientados a eventos que utilizan componentes que facilitan mucho la tarea de desarrollar aplicaciones a los programadores que no tienen apenas experiencia en programación. Usando las interfaces gráficas y los eventos, es muy sencillo realizar los primeros programas.

Este tipo de lenguajes son típicos en *Windows* o en *Mac* y lo que hacen normalmente las aplicaciones que han arrancado es quedarse a la espera de las acciones del usuario. Estas acciones son eventos.

Lo habitual en este tipo de programas es que la mayor parte del tiempo estén a la espera de que ocurra un evento y respondiendo a dichos eventos.

El evento más común es la pulsación de un *click* con el botón del ratón. Por supuesto, hay muchos tipos de eventos. También son muy comunes las dobles pulsaciones de ratón (*el doble click*). mover el ratón de una

posición a otra, arrastrar un ícono por la pantalla, pulsar una tecla o una combinación de teclas, seleccionar una opción de un menú, escribir en una caja de texto...

Cada elemento de la pantalla tiene sus propios eventos y la propia ventana o formulario tiene los suyos propios.

Cuando se va a escribir una parte del código para un evento de un determinado elemento, se crea automáticamente la cabecera de la función donde se va a escribir el contenido y aparecen también los parámetros que recibe el evento y por los que después se puede preguntar.

Formularios

Dentro de la programación gráfica en el entorno *Windows*, la parte sobre la que se realiza la colocación de elementos son las ventanas o formularios.

El aspecto básico de un formulario cuando se crea dentro de un proyecto es el siguiente:



Habitualmente, una aplicación gráfica contiene múltiples formularios. Siempre debe tener un formulario principal, que será con el que arranca la aplicación, aunque también puede ser directamente una clase de arranque la que se encargue de lanzar el primer formulario que se debe abrir.

Por supuesto, unos formularios pueden ir llamando a otros y así ir cambiando el flujo de la programación.

4. Programación gráfica

Existe un tipo de formulario especial que es el formulario *MDI* (*Multiple Document Interface*), que permite contener otros formularios dentro de sí mismo. A los formularios que están dentro se les llama formularios hijos y son del tipo *MDIChild*. La manera de mostrar los formularios hijos dentro del formulario padre es totalmente configurable.

Todos los objetos tienen sus propiedades y los formularios no dejan de ser objetos. Aunque tienen muchísimas propiedades, habitualmente muchas de ellas se dejan con los valores que tienen por defecto. La siguiente es una lista de las propiedades más habituales que se van a utilizar en un formulario:

Propiedad	Descripción
<i>AcceptButton</i>	Determina el botón del formulario que se activa cuando se presiona la tecla <i>Enter</i> .
<i>AllowDrop</i>	Establece un valor que indica si el control puede aceptar datos arrastrados por el usuario.
<i>AutoSizeMode</i>	Determina si el formulario debe escalar automáticamente cuando cambian la resolución o el tamaño de las fuentes.
<i>AutoScroll</i>	Establece un valor que indica si las barras de desplazamiento aparecen automáticamente o no.
<i>AutoScrollMargin</i>	Determina el tamaño del margen con los controles cuando hay barras de desplazamiento.
<i>AutoScrollMinSize</i>	Tamaño mínimo para las barras de desplazamiento.
<i>AutoSize</i>	Determina si el formulario modifica automáticamente su contenido al cambiar de tamaño o no.
<i>AutoSizeMode</i>	Modo en el que el formulario cambia automáticamente de tamaño.
<i>BackColor</i>	Color de fondo del formulario.
<i>BackgroundImage</i>	Imagen de fondo del formulario.
<i>CancelButton</i>	Determina el botón del formulario que se activa cuando se presiona la tecla <i>Esc</i> .
<i>ContextMenuStrip</i>	Menú de contexto que debe aparecer cuando se pulsa el botón derecho del ratón.
<i>Cursor</i>	Cursor que debe aparecer cuando el ratón se mueve sobre el formulario.

Propiedad	Descripción
<i>Enabled</i>	Indica si el formulario está activo (responde a las interacciones) o no.
<i>Font</i>	Tipo de letra con el que se va a escribir en el formulario.
<i>Forecolor</i>	Color con el que se va a escribir y dibujar en el formulario.
<i>FormBorderStyle</i>	Estilo del borde del formulario (si es fijo, si se puede cambiar de tamaño, etc.).
<i>Icon</i>	Icono del formulario (sale en la parte superior del formulario, pero además, si es el formulario principal, es el que aparece por defecto como ícono de la aplicación).
<i>IsMdiChild</i>	Indica si el formulario es un <i>MDIChild</i> y se debe mostrar contenido dentro de un formulario <i>MDI</i> .
<i>IsMdiContainer</i>	Indica si el formulario es un <i>MDI</i> que va a contener otros formularios.
<i>KeyPreview</i>	Permite que al realizar una pulsación desde el teclado, el evento llegue antes al formulario que al objeto que tenga el foco en ese momento.
<i>Locked</i>	Indica si el formulario está bloqueado (no se puede ni mover ni cambiar de tamaño) o no.
<i>MaximizeBox</i>	Determina si se muestra o no el botón de maximizar la ventana.
<i>MaximumSize</i>	Es el máximo al que se va a poder ampliar el formulario cuando se cambie de tamaño.
<i>Menu</i>	Define el menú principal del formulario que se visualiza en la parte superior de la ventana.
<i>MinimizeBox</i>	Determina si se muestra o no el botón de minimizar la ventana.
<i>MinimumSize</i>	Es el mínimo al que se va a poder reducir el formulario cuando se cambie de tamaño.
<i>Opacity</i>	Porcentaje de opacidad del formulario.
<i>ShowIcon</i>	Determina si se debe visualizar o no el ícono en la barra del título.
<i>ShowInTaskbar</i>	Determina si se debe visualizar o no el formulario en la barra de tareas de Windows.
<i>Size</i>	Tamaño del formulario.

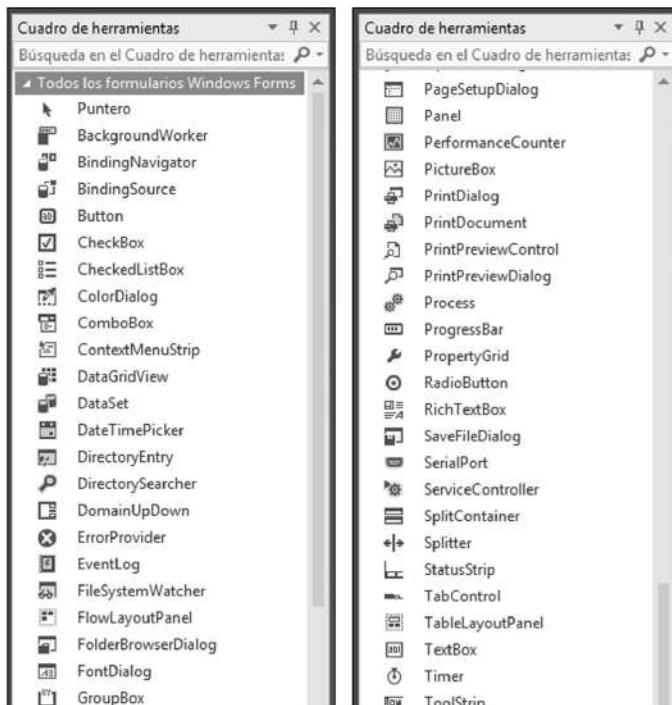
4. Programación gráfica

Propiedad	Descripción
<i>StartPosition</i>	Coordenadas de posición inicial del formulario dentro de la pantalla de Windows.
<i>Text</i>	Texto asociado al formulario.
<i>TopMost</i>	Determina si el formulario se debe colocar sobre el resto de formularios o no.
<i>TransparencyKey</i>	Es el color que van a representar las áreas transparentes del formulario.
<i>WindowState</i>	Estado inicial del formulario (minimizado, maximizado o normal).

Herramientas

La programación gráfica en *Visual C#* permite el uso de muchas herramientas que se pueden colocar en los formularios. Las herramientas que se van a utilizar están en el *Cuadro de herramientas* que se puede ver en la parte izquierda de las pantallas de desarrollo.

En las siguientes imágenes se pueden ver las herramientas disponibles:



Las herramientas se dividen en varios grupos dependiendo de su funcionalidad. En los siguientes apartados se van a ver las diferentes herramientas disponibles agrupadas por temáticas.

Controles comunes

La lista de controles comunes que tenemos a nuestra disposición dentro del cuadro de herramientas es la que se encuentra en la imagen siguiente:



Lo mejor es ir viendo los controles más comunes aplicados en diferentes ejemplos. Viendo el diseño de las ventanas y el código que se genera, se puede ir aprendiendo sobre la marcha de manera bastante sencilla.

El primer ejemplo va a consistir en realizar una aplicación en la que aparezca un texto de prueba, cuyas características se puedan modificar utilizando diferentes botones. Las posibles modificaciones serán:

- Cambiar los colores del texto.
- Cambiar la fuente de la letra.
- Cambiar el tamaño de la letra

4. Programación gráfica

- Cambiar el modo del texto (en negrita, en cursiva, tachado, subrayado).

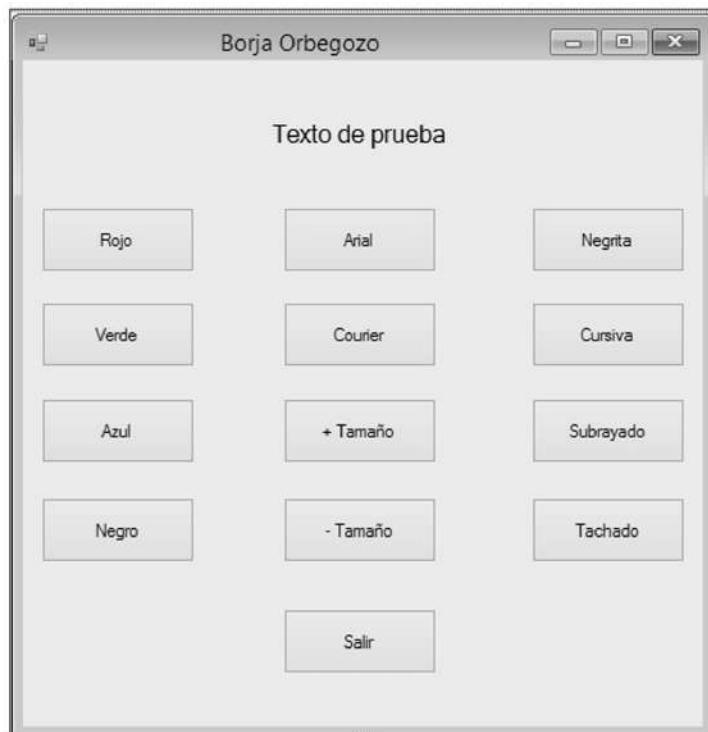
También se va a colocar un botón que permita cerrar la aplicación.

Este programa tiene varios objetivos. Por un lado, aprender a distinguir las propiedades de los métodos y de los objetos. Por otro lado, aprender la nomenclatura y los valores aceptados por los diferentes tipos de propiedades. Y, por supuesto, aprender a utilizar las primeras herramientas.

Los tipos de herramientas que se van a utilizar son dos:

- *Label*: etiquetas que contienen texto fijo.
- *Button*: botones de comando.

Lo primero que hay que hacer es crear el formulario con los elementos que van a ser necesarios para que la aplicación funcione. El aspecto del formulario debe ser como el de la siguiente imagen.



Ya que es el primer ejemplo, es preferible explicar paso a paso qué hay que hacer para que el texto cambie de color, letra, modo, etc.

Empezaremos por los botones de los colores. Antes de nada hay que aclarar cómo se puede poner cualquier color en un elemento de una ventana. Los colores utilizan el formato *RGB (Red, Green, Blue)*, que es la mezcla de colores rojo, verde y azul que se usa, por ejemplo, en televisión. Los valores que se pueden poner a cada uno de los colores van de 0 hasta 255. El valor 0 indica la ausencia de luz de ese color y el valor 255 indica el máximo de luz de ese color. Siendo así, las combinaciones de colores básicas son las siguientes:

- Negro: *RGB(0, 0, 0)*.
- Blanco: *RGB(255, 255, 255)*.
- Rojo: *RGB(255, 0, 0)*.
- Verde: *RGB(0, 255, 0)*.
- Azul: *RGB(0, 0, 255)*.

Para el resto de elementos no hay mucho más que explicar. Lo mejor es ver el código comentado para entender claramente cómo se puede realizar este programa.

Una recomendación importante en todos los programas es llamar a los elementos con nombres descriptivos. Microsoft recomienda usar 3 letras que describen el tipo de elemento. La siguiente es una lista con los elementos más comunes:

Control	Prefijo
<i>Label</i>	<i>lbl</i>
<i>TextBox</i>	<i>txt</i>
<i>ListBox</i>	<i>lsb</i>
<i>CheckBox</i>	<i>chk</i>
<i>RadioButton</i>	<i>rbt</i>
<i>Button</i>	<i>btn</i>
<i>Command</i>	<i>cmd</i>
<i>DataView</i>	<i>dvw</i>
<i>DataBase</i>	<i>db</i>
<i>DataTable</i>	<i>tbl</i>
<i> DataColumn</i>	<i>dtc</i>
...	...

4. Programación gráfica

Control	Prefijo
<i>DataSet</i>	<i>dst</i>
<i>DataReader</i>	<i>dr</i>
<i>Connection</i>	<i>con</i>
<i>Form</i>	<i>frm</i>

Aunque no es obligatorio utilizar esta nomenclatura, sí que es muy recomendable para aumentar la legibilidad del código. Si un desarrollador que debe modificar una aplicación desarrollada por otra persona ve en el código un objeto llamado *lblTexto* y otro llamado *btnSalir*, enseguida va a deducir que el primero es una etiqueta que contiene texto y que el segundo es un botón de comando que va a permitir salir de la ventana o del programa.

En el siguiente código se utiliza la misma nomenclatura para llamar a los botones, la etiqueta y el formulario:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Ejercicio01
{
    public partial class frmEjercicio01 : Form
    {
        public frmEjercicio01()
        {
            InitializeComponent();
        }

        private void butSalir_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void butRojo_Click(object sender, EventArgs e)
        {
            // Se pone el color rojo
            lblTexto.ForeColor = System.Drawing.Color.Fro-
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
}

private void butVerde_Click(object sender, EventArgs e)
{
    // Se pone el color verde
    lblTexto.ForeColor = System.Drawing.Color.FromArgb(0,
    255, 0);
}

private void butAzul_Click(object sender, EventArgs e)
{
    // Se pone el color azul
    lblTexto.ForeColor = System.Drawing.Color.FromArgb(0,
    0, 255);
}

private void butNegro_Click(object sender, EventArgs e)
{
    // Se pone el color negro
    lblTexto.ForeColor = System.Drawing.Color.FromArgb(0,
    0, 0);
}

private void butArial_Click(object sender, EventArgs e)
{
    // Se pone el tipo de letra Arial
    lblTexto.Font = new Font("Arial", lblTexto.Font.Size,
    lblTexto.Font.Style);
}

private void butCourier_Click(object sender, EventArgs e)
{
    // Se pone el tipo de letra Courier
    lblTexto.Font = new Font("Courier", lblTexto.Font.
    Size, lblTexto.Font.Style);
}

private void butMasTamano_Click(object sender, EventArgs
    e)
{
    // Se incrementa el tamaño de letra
    lblTexto.Font = new Font(lblTexto.Font.Name, lblTex-
    to.Font.Size+1, lblTexto.Font.Style);
}
```

4. Programación gráfica

```
private void butMenosTamano_Click(object sender, EventArgs e)
{
    // Se incrementa el tamaño de letra
    lblTexto.Font = new Font(lblTexto.Font.Name, lblTexto.Font.Size-1, lblTexto.Font.Style);
}

private void butNegrita_Click(object sender, EventArgs e)
{
    // Se pone letra negrita
    lblTexto.Font = new Font(lblTexto.Font.Name, lblTexto.Font.Size, FontStyle.Bold);
}

private void butCursiva_Click(object sender, EventArgs e)
{
    // Se pone letra cursiva
    lblTexto.Font = new Font(lblTexto.Font.Name, lblTexto.Font.Size, FontStyle.Italic);
}

private void butSubrayado_Click(object sender, EventArgs e)
{
    // Se pone letra negrita
    lblTexto.Font = new Font(lblTexto.Font.Name, lblTexto.Font.Size, FontStyle.Underline);
}

private void butTachado_Click(object sender, EventArgs e)
{
    // Se pone letra negrita
    lblTexto.Font = new Font(lblTexto.Font.Name, lblTexto.Font.Size, FontStyle.Strikeout);
}
```

El segundo ejemplo consiste en crear un programa que permita elegir una serie de cajas de verificación y que, después de pulsar un botón de comando, escriba en una etiqueta una frase que contenga los elementos que han sido seleccionados.

También se va a colocar un botón que permita cerrar la aplicación.

Este programa tiene varios objetivos. Por un lado, aprender a utilizar las casillas de verificación. Por el otro, aprender a declarar variables y a concatenar elementos de texto por partes.

En este caso, los tipos de herramientas que se van a utilizar son tres:

- *CheckBox*: cajas de verificación o chequeo.
- *Label*: etiquetas que contienen texto fijo.
- *Button*: botones de comando.



En este ejemplo lo ideal es crear una variable de tipo cadena que se vaya modificando en función de las opciones marcadas. Cuando ya se tenga el texto completo, entonces se asignará a la etiqueta de pantalla.

Para poder concatenar textos hay que usar el símbolo +.

En este ejemplo hay que tener en cuenta la cantidad de respuestas marcadas, lo cual se puede hacer con instrucciones *if* de control de flujo.

En cada ejemplo se va a procurar no repetir lo que ya está en ejercicios anteriores. Solamente se va a poner la parte interesante y novedosa con respecto a lo ya visto antes. En este caso, lo que interesa es ver el código del botón que realiza la selección y escribe la etiqueta. Dicho código puede

4. Programación gráfica

```
private void butSeleccionar_Click(object sender, EventArgs e)
{
    // Se declara la variable
    String sTexto = "Mi ordenador no tiene nada";
    // Se chequea si hay algún elemento seleccionado
    if (chkDVDROM.Checked || chkTarjetaSonido.Checked
        || chkTarjetaVideo.Checked || chkConexionUSB.Checked)
    {
        sTexto = "Mi ordenador tiene ";
        if (chkDVDROM.Checked)
        {
            sTexto = sTexto + "DVD-ROM";
            if (chkTarjetaSonido.Checked)
            {
                if (chkTarjetaVideo.Checked || chkConexionUSB.
                    Checked)
                {
                    sTexto = sTexto + ", Tarjeta de sonido";
                    if (chkTarjetaVideo.Checked)
                    {
                        if (chkConexionUSB.Checked)
                        {
                            sTexto = sTexto +
                                ", Tarjeta de video y Conexión USB";
                        }
                        else
                        {
                            sTexto = sTexto
                                + " y Tarjeta de video";
                        }
                    }
                    else
                    {
                        if (chkConexionUSB.Checked)
                        {
                            sTexto = sTexto + " y Conexión USB";
                        }
                    }
                }
            }
            else
            {
                sTexto = sTexto + " y Tarjeta de sonido";
            }
        }
        else
        {
            if (chkTarjetaVideo.Checked)
            {
                if (chkConexionUSB.Checked)
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
        sTexto = sTexto +
            ", Tarjeta de video y Conexión USB";
    }
    else
    {
        sTexto = sTexto
            + " y Tarjeta de video";
    }
}
else
{
    if (chkConexionUSB.Checked)
    {
        sTexto = sTexto + " y Conexión USB";
    }
}
else
{
    if (chkTarjetaSonido.Checked)
    {
        sTexto = sTexto + "Tarjeta de sonido";
        if (chkTarjetaVideo.Checked)
        {
            if (chkConexionUSB.Checked)
            {
                sTexto = sTexto
                    + "Tarjeta de video y Conexión USB";
            }
            else
            {
                sTexto = sTexto
                    + " y Tarjeta de video";
            }
        }
        else
        {
            if (chkConexionUSB.Checked)
            {
                sTexto = sTexto + " y Conexión USB";
            }
        }
    }
}
```

4. Programación gráfica

```
        }
    else
    {
        if (chkTarjetaVideo.Checked)
        {
            sTexto = sTexto + " Tarjeta de video";
            if (chkConexionUSB.Checked)
            {
                sTexto = sTexto + " y Conexión USB";
            }
        }
        else
        {
            if (chkConexionUSB.Checked)
            {
                sTexto = sTexto + " Conexión USB";
            }
        }
    }
}
// Se escribe el resultado en la etiqueta de pantalla
lblTexto.Text = sTexto;
}
```

El tercer ejemplo consiste en tener un grupo de botones de opción, de forma que se permita elegir una única opción de entre las posibles. De la misma forma que en el ejemplo anterior, cuando se pulse el botón correspondiente, se escribirá en una etiqueta de texto la opción que se ha escogido.

El objetivo de este programa es conocer el funcionamiento de los botones de opción de manera sencilla.

En este caso, los tipos de herramientas que se van a utilizar son tres:

- *RadioButton*: botones de opción.
- *Label*: etiquetas que contienen texto fijo.
- *Button*: botones de comando.



En este ejemplo no es necesario disponer de una variable auxiliar, puesto que es muy sencillo construir el texto de la etiqueta. Dicho texto va a tener una parte fija y otra parte que va a depender de lo que haya chequeado en las opciones.

En este ejemplo se puede poner el texto utilizando instrucciones *if* de control de flujo, de manera similar al ejercicio anterior, pero sin tantas combinaciones.

En este caso, como en el anterior, lo que interesa es ver el código del botón que realiza la selección y escribe la etiqueta. Dicho código puede ser como el siguiente:

```
private void butSeleccionar_Click(object sender, EventArgs  
e)  
{  
    // Primero se pone el texto fijo  
    lblTexto.Text =  
        "Mi aparato de entretenimiento preferido es: ";  
    // Se chequea el elemento que esté seleccionado  
    if (radMovil.Checked)  
    {
```

4. Programación gráfica

```
// Se escribe el resultado en la etiqueta
lblTexto.Text = lblTexto.Text + "Móvil";
}
else
{
    if (radOrdenador.Checked)
    {
        // Se escribe el resultado en la etiqueta
        lblTexto.Text = lblTexto.Text + "Ordenador";
    }
    else
    {
        if (radTablet.Checked)
        {
            // Se escribe el resultado en la etiqueta
            lblTexto.Text = lblTexto.Text + "Tablet";
        }
        else
        {
            // Sólo puede ser el elemento restante
            // Se escribe el resultado en la etiqueta
            lblTexto.Text = lblTexto.Text + "Televisión";
        }
    }
}
}
```

En este ejemplo, dado que lo que se muestra al final de la etiqueta es lo mismo que aparece en las opciones, se podrían haber utilizado esos valores. La ventaja que tiene es que si más adelante se va a modificar alguno de los contenidos, no será necesario modificar el código. El mismo código utilizando estas opciones sería como el siguiente:

```
private void butSeleccionar_Click(object sender, EventArgs e)
{
    // Primero se pone el texto fijo
    lblTexto.Text =
        "Mi aparato de entretenimiento preferido es: ";
    // Se chequea el elemento que esté seleccionado
    if (radMovil.Checked)
    {
        // Se escribe el resultado en la etiqueta
        lblTexto.Text = lblTexto.Text + radMovil.Text;
    }
    else
    {
        if (radOrdenador.Checked)
    }
}
```

```
// Se escribe el resultado en la etiqueta  
lblTexto.Text = lblTexto.Text + radOrdenador.  
Text;  
}  
else  
{  
    if (radTablet.Checked)  
    {  
        // Se escribe el resultado en la etiqueta  
        lblTexto.Text = lblTexto.Text + radTablet.  
        Text;  
    }  
    else  
    {  
        // Sólo puede ser el elemento restante  
        // Se escribe el resultado en la etiqueta  
        lblTexto.Text = lblTexto.Text + radTelevi-  
        sion.Text;  
    }  
}  
}  
}
```

En el cuarto ejemplo se va a controlar la pulsación de los diferentes botones del ratón. Es importante controlar los diferentes botones, que después implicarán diferentes acciones en el uso de las aplicaciones. En este ejemplo se va a tener una etiqueta en la ventana en la que, en función del botón pulsado, aparecerá uno de los siguientes mensajes:

- Pulsado el botón izquierdo del ratón.
- Pulsado el botón derecho del ratón.
- Pulsado el botón central del ratón.

Este ejemplo persigue varios objetivos. Por un lado, aprender a controlar las pulsaciones del ratón y saber qué botón es el que ha sido pulsado. Por otro lado, distinguir entre las pulsaciones en un objeto y las pulsaciones en el fondo de la ventana.

En este caso, la única herramienta que se va a utilizar es:

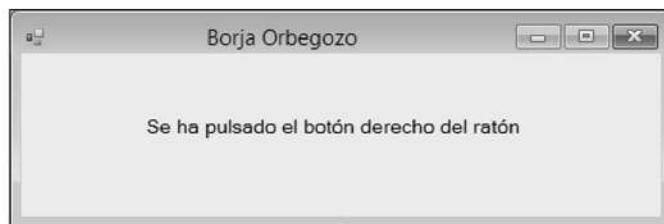
- *Label*: etiquetas que contienen texto fijo.

Hay que tener en cuenta que la pulsación del ratón en este ejemplo se puede producir no solamente en el propio formulario, sino también sobre

4. Programación gráfica

la propia etiqueta. Si se produce sobre la etiqueta, el evento será asociado al objeto etiqueta y por ello hay que duplicar el código.

La ventana de la aplicación va a tener el siguiente aspecto:



Aunque el evento habitual para controlar la pulsación de un ratón es el evento *Click*, en dicho evento no se puede controlar qué botón ha sido pulsado. Para este caso el evento que se debe utilizar es el *MouseClick*. En este evento se recibe como parámetro un evento que devuelve el botón pulsado.

El código necesario para resolver el ejemplo, duplicando el código de los eventos, puede ser como el siguiente:

```
private void frmEjercicio04_MouseClick(object sender, MouseEventArgs e)
{
    // Se pregunta por el botón que ha sido pulsado
    if (e.Button.ToString() == "Left")
    {
        lblTexto.Text =
            "Pulsado el botón izquierdo del ratón";
    }
    else
    {
        if (e.Button.ToString() == "Right")
        {
            lblTexto.Text =
                "Pulsado el botón derecho del ratón";
        }
        else
        {
            lblTexto.Text =
                "Pulsado el botón central del ratón";
        }
    }
}
```

```
private void lblTexto_MouseClick(object sender,
    MouseEventArgs e)
{
    // Se pregunta por el botón que ha sido pulsado
    if (e.Button.ToString() == "Left")
    {
        lblTexto.Text =
            "Pulsado el botón izquierdo del ratón";
    }
    else
    {
        if (e.Button.ToString() == "Right")
        {
            lblTexto.Text =
                "Pulsado el botón derecho del ratón";
        }
        else
        {
            lblTexto.Text =
                "Pulsado el botón central del ratón";
        }
    }
}
```

Por supuesto, este código es muy optimizable, porque no tiene sentido copiar y pegar el mismo código en más de una función. Lo que se debe hacer es llamar de un evento a otro, puesto que lo que se va a ejecutar es lo mismo. En este caso, el código para el evento de pulsación del ratón en la etiqueta simplemente debe llamar al mismo evento del formulario, pasando los mismos parámetros que ha recibido de entrada. Quedaría así:

```
private void lblTexto_MouseClick(object sender,
    MouseEventArgs e)
{
    // Se llama al mismo evento del formulario
    frmEjercicio04_MouseClick(sender, e);
}
```

No solamente se simplifica el código, sino que además, si hay que realizar alguna modificación, sólo se debe hacer una vez y no hay que duplicar el cambio en el contenido del otro evento.

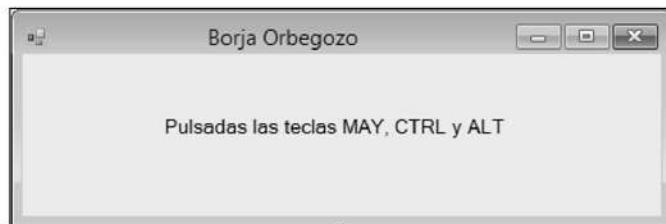
4. Programación gráfica

En el **quinto ejemplo** se va a hacer algo similar, pero controlando las pulsaciones de las teclas especiales del teclado. En la ventana también habrá una única etiqueta y en función de las teclas pulsadas aparecerá uno de los siguientes mensajes:

- Pulsada la tecla *MAY*.
- Pulsada la tecla *CTRL*.
- Pulsada la tecla *ALT*.
- Pulsadas las teclas *MAY* y *CTRL*.
- Pulsadas las teclas *MAY* y *ALT*.
- Pulsadas las teclas *CTRL* y *ALT*.
- Pulsadas las teclas *MAY*, *CTRL* y *ALT*.

El principal objetivo de este ejemplo es el control de los eventos de pulsación de teclas. También se va a repasar el control de flujo con las diferentes posibles combinaciones de teclas.

El aspecto del formulario debe ser como el de la siguiente imagen.



Cuando un formulario tiene una serie de objetos, siempre suele haber uno de ellos que es el activo. Cuando se produce una pulsación de una tecla, el objeto que recoge dicha pulsación es el activo en ese momento. Si lo que se desea es controlar esa pulsación de forma genérica, lo que hay que hacer es que el propio formulario sea quien recoja la pulsación, de forma que el evento del propio formulario sea el que tenga que realizar el tratamiento de la tecla pulsada.

La propiedad que le indica al formulario que debe recoger los eventos de pulsación del teclado es *KeyPreview*. Si dicha propiedad está designada como *false*, cada objeto activo recoge directamente las pulsaciones. Si está designada como *true*, entonces el formulario es el que recibe el evento de

El código fuente del evento del formulario que controla estas teclas especiales debe ser como el siguiente:

```
private void frmEjercicio05_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Shift)
    {
        // Se ha pulsado la tecla MAY
        if (e.Control)
        {
            // Se ha pulsado la tecla CTRL
            if (e.Alt)
            {
                // Se ha pulsado la tecla ALT
                lblTexto.Text =
                    "Pulsadas las teclas MAY, CTRL y ALT";
            }
            else
            {
                // No se ha pulsado la tecla ALT
                lblTexto.Text =
                    "Pulsadas las teclas MAY y CTRL";
            }
        }
        else
        {
            // No se ha pulsado la tecla CTRL
            if (e.Alt)
            {
                // Se ha pulsado la tecla ALT
                lblTexto.Text =
                    "Pulsadas las teclas MAY y ALT";
            }
            else
            {
                // No se ha pulsado la tecla ALT
                lblTexto.Text = "Pulsada la tecla MAY";
            }
        }
    }
    else
    {
        // No se ha pulsado la tecla MAY
```

4. Programación gráfica

```
if (e.Control)
{
    // Se ha pulsado la tecla CTRL
    if (e.Alt)
    {
        // Se ha pulsado la tecla ALT
        lblTexto.Text =
            "Pulsadas las teclas CTRL y ALT";
    }
    else
    {
        // No se ha pulsado la tecla ALT
        lblTexto.Text = "Pulsada la tecla CTRL";
    }
}
else
{
    // No se ha pulsado la tecla CTRL
    if (e.Alt)
    {
        // Se ha pulsado la tecla ALT
        lblTexto.Text = "Pulsada la tecla ALT";
    }
    else
    {
        // No se ha pulsado la tecla ALT
        lblTexto.Text = "";
    }
}
```

Como se puede apreciar en el código, el evento de la pulsación que se ha utilizado ha sido *KeyDown*, que marca cuando una tecla se pulsa hacia abajo. También se podría haber utilizado *KeyUp*, que marca cuando se suelta la tecla. No se podría haber utilizado el evento *KeyPress*, porque en dicho evento no se controlan las teclas especiales del teclado.

El parámetro *e* que se recibe en estos dos eventos permite saber si se han pulsado o no las teclas especiales. Se puede preguntar si son *true* o *false* con los siguientes elementos:

- *e.Shift* → pulsada la tecla MAY.
- *e.Control* → pulsada la tecla CTRL.
- *e.Alt* → pulsada la tecla ALT.

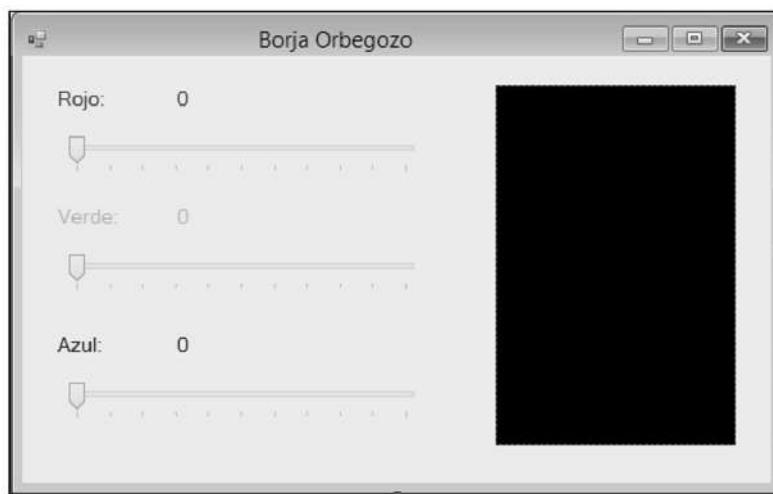
El sexto ejemplo consiste en controlar las barras de desplazamiento. Se van a establecer unos límites de valores de 0 a 255 y mediante tres barras se controlará el color de fondo de un objeto de imagen. Además, los valores de las barras de desplazamiento aparecerán en unas etiquetas, para que se vea claramente lo que se está seleccionando.

El objetivo del ejemplo es aprender a utilizar las barras de desplazamiento y conocer el objeto para guardar imágenes. También se va a reforzar el uso de las mezclas de colores mediante la función correspondiente.

En este caso, los tipos de herramientas que se van a utilizar son tres:

- *Label*: etiquetas que contienen texto fijo o variable desde código.
- *TrackBar*: barras de desplazamiento.
- *PictureBox*: control de imagen.

El aspecto del formulario debe ser como el de la siguiente imagen.



En este ejemplo es importante tener las propiedades bien iniciadas en tiempo de diseño, para que después en tiempo de ejecución todo funcione correctamente. En este caso, la clave está en la herramienta *TrackBar*, que no aparece en la lista de controles comunes, pero que sí se puede buscar en la lista general con todos los controles. Cada una de las barras debe estar inicializada con los valores mínimo y máximo tal y como se solicita. El valor mínimo será 0 y se debe colocar en la propiedad *Minimum*. El

4. Programación gráfica

Cada vez que se mueva una de las barras, habrá que cambiar la etiqueta correspondiente y después modificar el color de fondo de la imagen. Aunque lo habitual es utilizar este objeto con una fotografía o un ícono, en este caso se va a tener un recuadro sin nada y se va a jugar con la propiedad del color de fondo (*BackColor*).

Una vez especificados los términos, lo mejor es ver el código de programación y tener en cuenta los comentarios.

```
private void traRojo_Scroll(object sender, EventArgs e)
{
    // Se escribe el valor del color rojo
    lblRojoDatos.Text = traRojo.Value.ToString();
    // Se cambia el color de la mezcla
    picColor.BackColor = Color.FromArgb(traRojo.Value, tra-
        Verde.Value, traAzul.Value);
}

private void traVerde_Scroll(object sender, EventArgs e)
{
    // Se escribe el valor del color verde
    lblVerdeDatos.Text = traVerde.Value.ToString();
    // Se cambia el color de la mezcla
    picColor.BackColor = Color.FromArgb(traRojo.Value, tra-
        Verde.Value, traAzul.Value);
}

private void traAzul_Scroll(object sender, EventArgs e)
{
    // Se escribe el valor del color azul
    lblAzulDatos.Text = traAzul.Value.ToString();
    // Se cambia el color de la mezcla
    picColor.BackColor = Color.FromArgb(traRojo.Value, tra-
        Verde.Value, traAzul.Value);
}
```

Como se puede apreciar en el código, el evento escogido para realizar las acciones es el evento *Scroll* de las *Trackbars*. Este evento salta cada vez que se mueve el indicador, de manera que se pueden conseguir los valores de uno en uno. Resulta imprescindible en este caso, ya que de esta forma se puede apreciar al momento el cambio del color según se van moviendo las barras de desplazamiento.

El séptimo ejemplo va a consistir en gestionar una lista de elementos. Se van a tener varios botones que permitan añadir nuevos elementos a la lista, eliminar un elemento de la lista o borrar la lista completa. Además, debe haber una caja de texto en la que se pueda escribir el nuevo elemento que se va a añadir. También debe haber un botón para salir de la aplicación.

Las acciones de los diferentes botones van a ser:

- Añadir:
 - Añade el contenido de la caja de texto al final de la lista.
 - Elimina el contenido de la caja de texto.
- Eliminar:
 - Busca el elemento que está seleccionado en la lista.
 - Elimina el elemento de la lista.
- Borrar todos:
 - Vacía la lista.

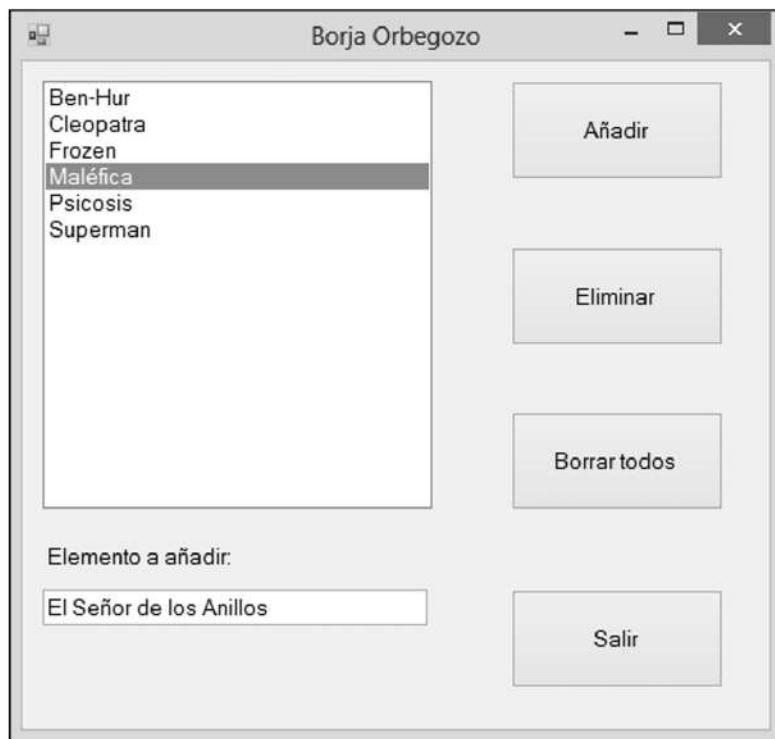
Los objetivos del ejemplo son varios. De entrada, aprender a utilizar las cajas de texto, que son bastante similares a las etiquetas, pero modificables. Por otra parte, el uso de las listas, aprendiendo a añadir y eliminar elementos.

En este caso, los tipos de herramientas que se van a utilizar son cuatro:

- *Label*: etiquetas que contienen texto fijo o variable desde código.
- *ListBox*: listas de elementos.
- *TextBox*: cajas de texto.
- *Button*: botones de comando.

El aspecto del formulario debe ser similar al de la siguiente imagen.

4. Programación gráfica



En este caso, la lista puede tener una serie de elementos predefinidos que se pueden colocar directamente en las propiedades o rellenarse cuando la ventana se cargue por primera vez. Para ello se debe utilizar el evento *Load* del formulario principal.

El resto de temas que debemos tener en cuenta se pueden ver directamente en el código comentado:

```
private void frmEjercicio07_Load(object sender, EventArgs e)
{
    // Al arrancar, se llenan los primeros elementos
    lstPeliculas.Items.Clear();
    lstPeliculas.Items.Add("Ben-Hur");
    lstPeliculas.Items.Add("Cleopatra");
    lstPeliculas.Items.Add("Frozen");
    lstPeliculas.Items.Add("Maléfica");
    lstPeliculas.Items.Add("Psicosis");
    lstPeliculas.Items.Add("Superman");
}

private void butAnadir_Click(object sender, EventArgs e)
```

```
// Sólo se añade si hay algún elemento
if (txtElemento.Text.Length>0)
{
    // Nuevo elemento para añadir a la lista
    lstPeliculas.Items.Add(txtElemento.Text);
    // Se vacía la caja de texto
    txtElemento.Text = "";
}
}

private void butEliminar_Click(object sender, EventArgs e)
{
    // Primero se verifica si hay algún elemento seleccionado
    if (lstPeliculas.SelectedItems.Count>0)
    {
        // El elemento que se elimina es el seleccionado
        lstPeliculas.Items.RemoveAt(
            lstPeliculas.SelectedIndices[0]);
    }
}

private void butBorrarTodos_Click(object sender, EventArgs e)
{
    // Se vacía la lista
    lstPeliculas.Items.Clear();
}
```

Como se puede ver en el código, hay un par de controles necesarios para que la aplicación tenga sentido. El primero está en el botón de añadir elemento, que debe testear antes de nada si en la caja de texto hay algo escrito. Para ello se utiliza *Length*, que devuelve el tamaño que tiene el contenido de la caja de texto y que está en la propiedad *Text*.

El otro control es el que se hace al eliminar, ya que hay que verificar primero si alguno de los elementos de la lista está seleccionado. Por defecto la lista no tiene ningún elemento seleccionado. En este caso se utiliza *Count*, que indica el número de elementos para listas o *arrays*.

El octavo y último ejemplo de este grupo es similar al anterior, pero utilizando cajas combinadas. Estas cajas son una especie de mezcla entre las cajas de texto y las listas. El funcionamiento de la parte de la lista es muy similar al de las propias listas. La parte de texto, en principio, se puede asociar a la lista o no. Si se elige un elemento de la lista, entonces aparece en la parte del texto. Pero también se puede modificar el texto de forma separada.

4. Programación gráfica

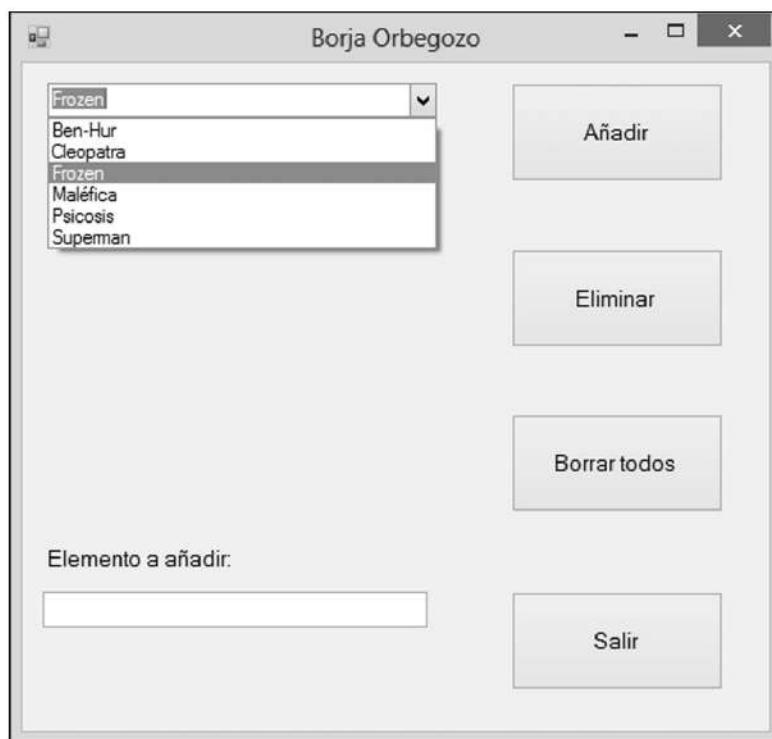
Se van a poner los mismos botones que en el ejemplo anterior, ejecutando también las mismas acciones.

El objetivo del ejemplo es aprender a utilizar las cajas combinadas añadiendo y eliminando elementos.

En este caso, los tipos de herramientas que se van a utilizar son cuatro:

- *Label*: etiquetas que contienen texto fijo o variable desde código.
- *ComboBox*: cajas combinadas.
- *TextBox*: cajas de texto.
- *Button*: botones de comando.

En el diseño se debe obtener una ventana como la de la siguiente imagen.



También en este ejemplo se van a llenar unos elementos iniciales cuando se lance la aplicación.

El código de la aplicación va a ser como el siguiente:

```
private void frmEjercicio08_Load(object sender, EventArgs e)
{
    // Al arrancar, se rellenan los primeros elementos
    cboPeliculas.Items.Clear();
    cboPeliculas.Items.Add("Ben-Hur");
    cboPeliculas.Items.Add("Cleopatra");
    cboPeliculas.Items.Add("Frozen");
    cboPeliculas.Items.Add("Maléfica");
    cboPeliculas.Items.Add("Psicosis");
    cboPeliculas.Items.Add("Superman");
}

private void butAnadir_Click(object sender, EventArgs e)
{
    // Sólo se añade si hay algún elemento
    if (txtElemento.Text.Length>0)
    {
        // Nuevo elemento para añadir a la lista
        cboPeliculas.Items.Add(txtElemento.Text);
        // Se vacía la caja de texto
        txtElemento.Text = "";
    }
}

private void butEliminar_Click(object sender, EventArgs e)
{
    // Primero se verifica si hay algún elemento seleccionado
    if (cboPeliculas.Text.Length > 0)
    {
        // El elemento que se elimina es el seleccionado
        cboPeliculas.Items.Remove(cboPeliculas.Text);
    }
}

private void butBorrarTodos_Click(object sender, EventArgs e)
{
    // Se vacía la lista
    cboPeliculas.Items.Clear();
}
```

4. Programación gráfica

Como se puede apreciar, el código es muy similar al del ejemplo anterior. La principal diferencia, aparte del uso del tipo diferente de herramienta, es el chequeo en el botón de eliminar. En las listas lo que se hacía era verificar si había algún elemento seleccionado. En este caso, cuando se selecciona un elemento, dicho elemento se coloca en la caja de texto de la propia caja combinada, así que lo que se hace es verificar directamente el contenido del texto de la caja combinada.

A la hora de eliminar, en lugar de usar el elemento por posición, se utiliza el método *Remove*, al que se le pasa el elemento que hay que eliminar. Dado que dicho elemento está en la parte del texto, se puede usar directamente.

Contenedores

La lista de contenedores a nuestra disposición en el cuadro de herramientas es la que se encuentra en la imagen siguiente.



En general, los contenedores permiten agrupar elementos de forma ordenada. Los paneles son los más sencillos y básicamente se utilizan para temas de diseño ordenado. Los demás son diferentes maneras de tener los controles mejor organizados y, en algunos casos, como los botones de radio, para separarlos por grupos.

Para ver cómo funcionan lo mejor es utilizar un par de ejemplos. Una vez más, con ejemplos que utilizan estas herramientas se va a ir aprendiendo sobre la marcha de manera sencilla.

El primero de los ejemplos va a utilizar la herramienta *GroupBox*, dentro de la cual se van a colocar varios botones de radio en forma de grupos. Se van a solicitar tres descripciones de una persona como son su color de pelo, su complejión y su deporte favorito. Cada una de las descripciones va a tener cuatro posibles respuestas, de forma que estarán agrupadas de cuatro en cuatro. Habrá una opción por defecto ya puesta cuando arranque la aplicación.

Cuando se pulse en un botón que indique que ya se ha hecho la selección, se va a escribir en una etiqueta el resultado de las opciones elegidas. Se debe hacer en varias frases que vayan concatenadas.

También se va a tener un botón que permita salir de la aplicación.

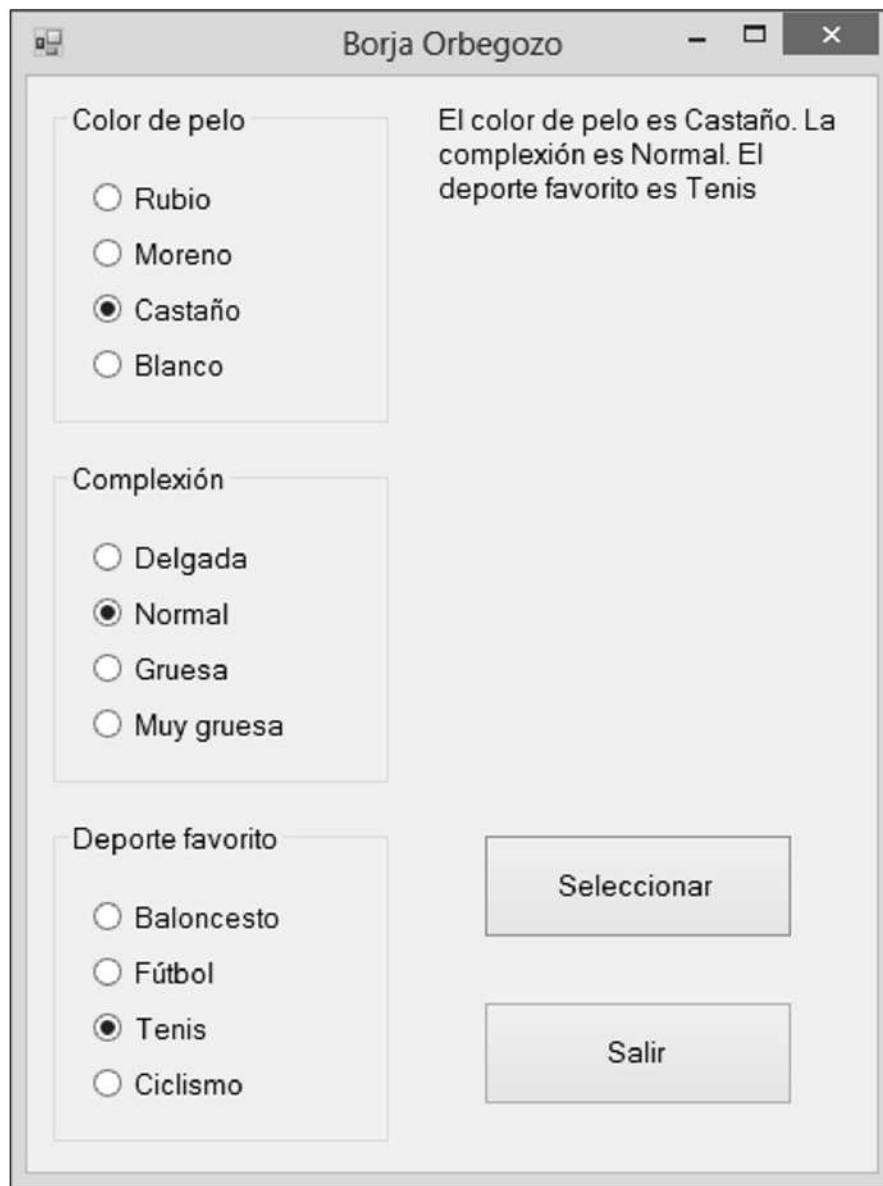
Este ejemplo va a tener varios objetivos. El primer objetivo es aprender a utilizar las cajas agrupadas, viendo cómo colocar botones de radio dentro de ellas de manera grupal. El otro, aprender a escribir etiquetas con varias líneas de texto, ya que por defecto una etiqueta únicamente ocupa una línea.

Para este ejemplo, los tipos de herramientas que se van a utilizar son cuatro:

- *Label*: etiquetas que contienen texto fijo o variable desde código.
- *GroupBox*: cajas agrupadas.
- *RadioButton*: botones de radio.
- *Button*: botones de comando.

En el diseño se debe obtener una ventana como la de la siguiente imagen.

4. Programación gráfica



Antes de ver el código, se van a comentar un par de cosas. Para que una etiqueta tenga varias líneas, existen varias opciones, que se van a describir en los siguientes puntos:

- Se pueden establecer exactamente los lugares en los que se desean los saltos de línea. Cada vez que se quiera poner un salto, se deberán poner explícitamente los comandos `\r\n`.

- Si se añade una arroba (@) delante de un texto entrecomillado, se divide el texto en varias líneas según la necesidad, utilizando los espacios como separadores cuando la siguiente palabra no entre en una línea.
- Cuando el texto está en una variable y se asigna directamente a la etiqueta, funciona como en el caso anterior.

Para este ejemplo se va a ir creando el texto en una variable, tal y como se ha hecho en ocasiones anteriores. Cuando ya se tenga todo, se asignará a la etiqueta de salida. El código debe ser como el siguiente:

```
private void butSeleccionar_Click(object sender, EventArgs
e)
{
    String sTexto = "";
    // Se va creando el texto de grupo en grupo de botones
    // Se busca el color del pelo
    if (radRubio.Checked)
    {
        sTexto = "El color de pelo es " + radRubio.Text +
        ". ";
    }
    else
    {
        if (radMoreno.Checked)
        {
            sTexto = "El color de pelo es " + radMoreno.Text
            + ". ";
        }
        else
        {
            if (radCastano.Checked)
            {
                sTexto = "El color de pelo es "
                + radCastano.Text + ". ";
            }
            else
            {
                sTexto = "El color de pelo es "
                + radBlanco.Text + ". ";
            }
        }
    }
    // Se busca la compleción
    if (radDelgada.Checked)
```

4. Programación gráfica

```
{  
    sTexto = sTexto + " La compleción es "  
        + radDelgada.Text + ". ";  
}  
else  
{  
    if (radNormal.Checked)  
    {  
        sTexto = sTexto + " La compleción es "  
            + radNormal.Text + ". ";  
    }  
    else  
{  
        if (radGruesa.Checked)  
        {  
            sTexto = sTexto + " La compleción es "  
                + radGruesa.Text + ". ";  
        }  
        else  
{  
            sTexto = sTexto + " La compleción es "  
                + radMuyGruesa.Text + ". ";  
        }  
    }  
}  
// Se busca el deporte favorito  
if (radBaloncesto.Checked)  
{  
    sTexto = sTexto + " El deporte favorito es "  
        + radBaloncesto.Text;  
}  
else  
{  
    if (radFutbol.Checked)  
    {  
        sTexto = sTexto + " El deporte favorito es "  
            + radFutbol.Text;  
    }  
    else  
{  
        if (radTenis.Checked)  
        {  
            sTexto = sTexto + " El deporte favorito es "  
                + radTenis.Text;  
        }  
        else  
{  
            sTexto = sTexto + " El deporte favorito es "  
                + radCiclismo.Text;  
        }  
    }  
}
```

```
        }
    }
}

// Finalmente se coloca el resultado en la etiqueta
lb1Resultado.Text = sTexto;
}
```

Como se puede apreciar en el código, lo único que hay que hacer es chequear los botones de radio en forma agrupada. No hay que preocuparse de si en un grupo hay alguno seleccionado o no, puesto que en diseño basta con poner uno por defecto en cada grupo y después no hay que hacer nada especial para que siempre haya un botón seleccionado por cada uno de los grupos.

En el siguiente ejemplo lo que se va a ver es el funcionamiento de las pestañas, que vienen muy bien para organizar los elementos de pantalla de forma agrupada. Y lo más positivo es que se pueden tener múltiples controles colocados en el mismo espacio de la ventana, pero como si fueran diferentes capas.

Lo que se va a hacer en este ejemplo es unir los tres primeros ejemplos que se han visto en el apartado de controles comunes, de manera que cada uno de ellos esté en una pestaña diferente.

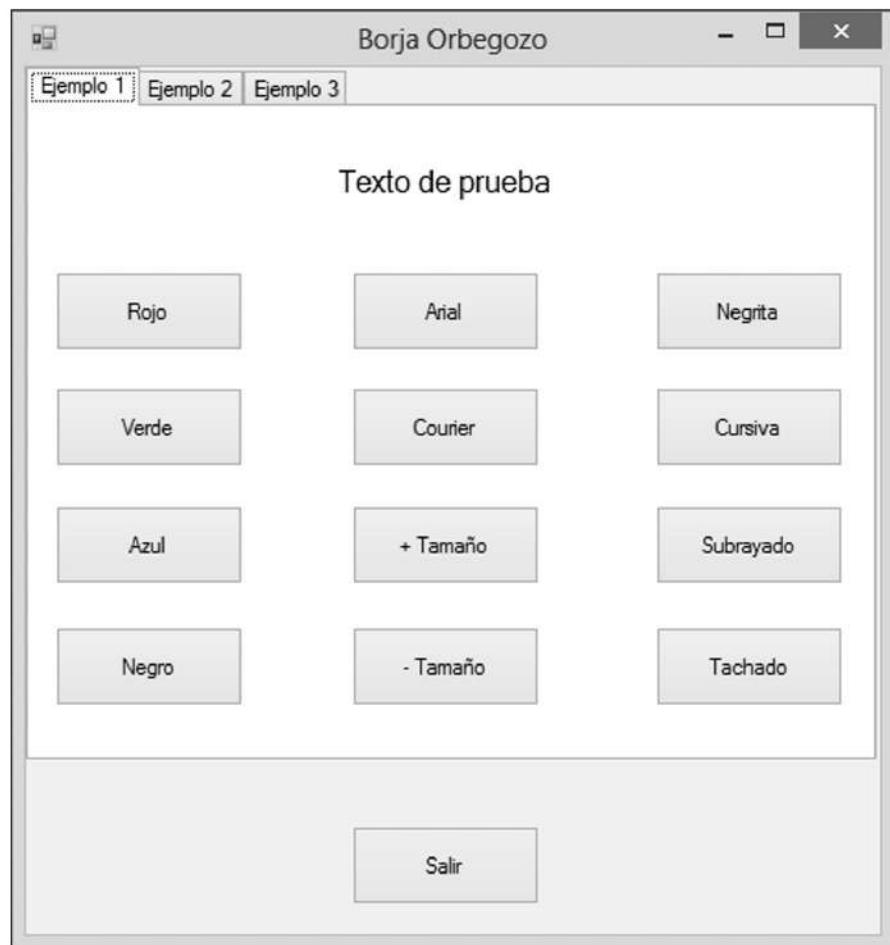
El objetivo de este ejemplo va a ser básicamente aprender a usar las pestañas como contenedores de herramientas.

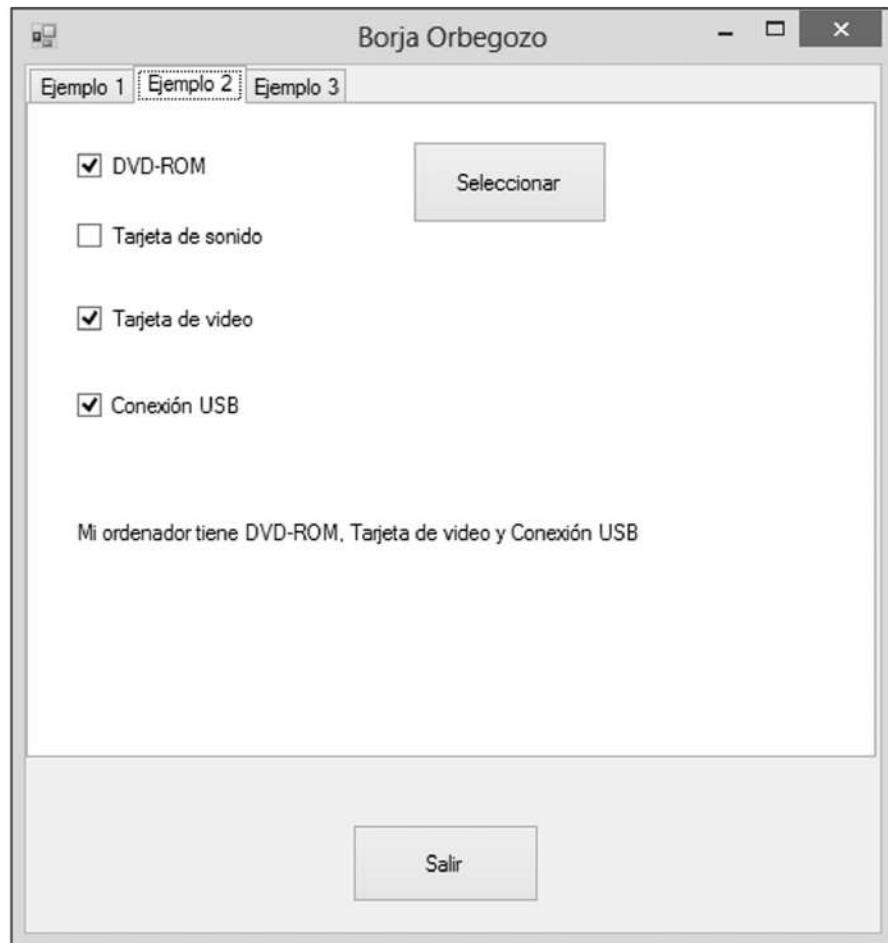
Para este ejemplo, el tipo de herramienta que se va a utilizar, aparte de las que ya se usaban en los otros ejemplos que lo integran, es:

- *TabControl*: control de tabulación.

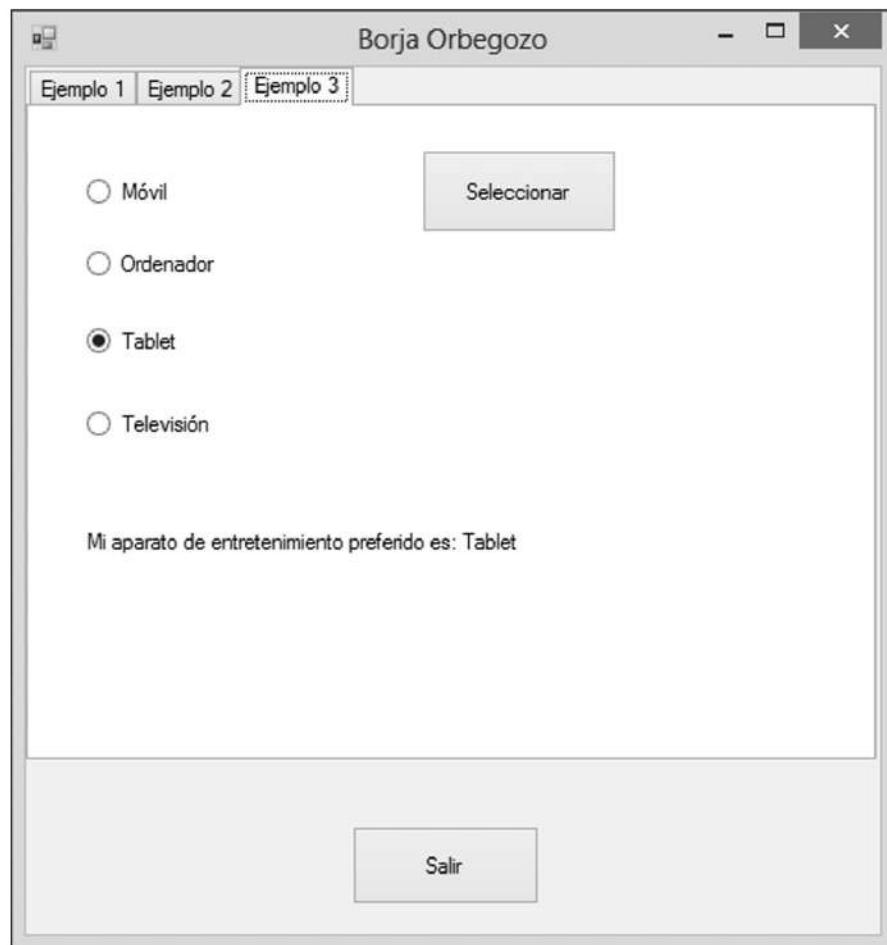
El diseño de esta ventana es el que se puede ver en las tres imágenes que se incluyen a continuación. Cada una de ellas corresponde a una pestaña diferente que se va a identificar con el ejemplo oportuno que se muestra seguidamente.

4. Programación gráfica





4. Programación gráfica



Como se puede apreciar en las imágenes anteriores, los controles de los diferentes ejemplos están posicionados en el mismo lugar de la ventana, pero en diferentes capas que están organizadas gracias a las pestañas superiores.

El código interno para cada una de las pestañas es el mismo que el de los ejemplos originales, así que basta con copiar y pegar todo el contenido. Lo único mencionable es que se ha quitado el botón *Sair* de la parte de las pestañas, puesto que así se tiene un botón común para todas.

Menús y barras de herramientas

La lista de menús y barras de herramientas que tenemos a nuestra disposición en el cuadro de herramientas es la que se encuentra en la imagen siguiente:



Los menús y submenús permiten agrupar en la parte superior de la ventana múltiples acciones que va a ejecutar la aplicación. Es una forma muy habitual, que se utiliza en la mayor parte de las aplicaciones.

También existen los menús contextuales, que se pueden asociar directamente a los controles de las ventanas. Cuando se pulsa con el botón derecho del ratón sobre un objeto que tiene un menú contextual asociado, aparece sobre la ventana el menú correspondiente y se pueden ejecutar las acciones de la misma forma que en los menús normales.

Las barras de herramientas son similares a los menús, pero con botones que suelen tener apariencia gráfica. Suelen ser una forma más visual de mostrar las acciones que se pueden ejecutar en una aplicación.

Suele ser una práctica muy habitual utilizar menús y barras de herramientas de forma combinada, incluso duplicando las acciones que muestran. Lo normal es que el menú contenga todas las acciones disponibles y que la barra de herramientas contenga sólo las más habituales, para que se pueda pulsar cómodamente sobre ellas en lugar de tener que navegar por los menús y submenús.

De la misma forma que con las otras herramientas, lo mejor es ir viendo algunos ejemplos para explicar cómo se diseñan este tipo de aplicaciones.

--- 4. Programación gráfica ---

El primer ejemplo es una aplicación muy simple que va a tener en la ventana una etiqueta y una imagen, las cuales van a ser modificadas desde un menú.

El menú principal va a contar con las siguientes opciones y subopciones:

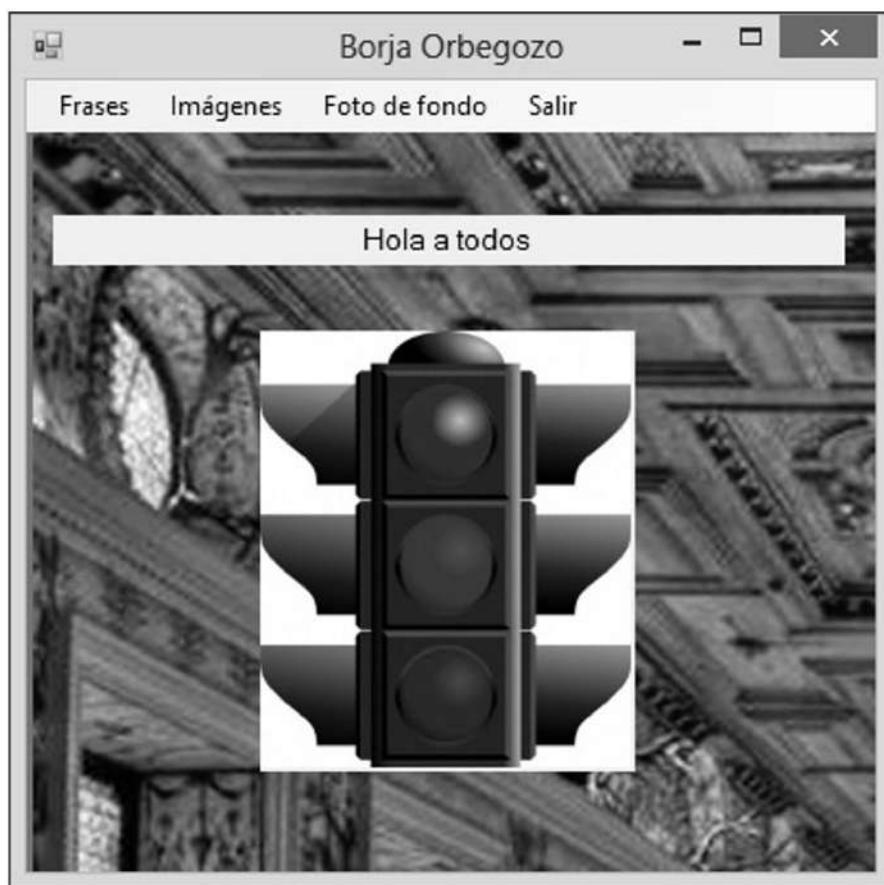
- Frases:
 - Saludo → escribe un saludo en la etiqueta.
 - Despedida → escribe una despedida en la etiqueta.
 - Nombre de pila → escribe un nombre de pila en la etiqueta.
- Imágenes:
 - Verde → cambia la imagen por un semáforo en verde.
 - Ámbar → cambia la imagen por un semáforo en ámbar.
 - Rojo → cambia la imagen por un semáforo en rojo.
- Foto de fondo:
 - Imagen 1 → cambia la imagen de fondo por una cualquiera.
 - Imagen 2 → cambia la imagen de fondo por otra diferente.
- Salir: finaliza la aplicación.

Este ejemplo tiene varios objetivos. El principal es el aprendizaje de los menús y submenús. Aparte de eso, se va a aprender a modificar imágenes de objetos y del fondo del formulario.

Para este ejemplo, los tipos de herramienta que se van a utilizar son tres:

- *Label*: etiquetas que contienen texto fijo o variable desde código.
- *PictureBox*: controles de imagen.
- *MenuStrip*: menús principales.

En el diseño se debe obtener una ventana como la de la imagen siguiente:



Este ejemplo tiene mucha parte de diseño, ya que los menús se pueden realizar directamente en tiempo de diseño. A los menús y submenús se les van a colocar directamente los nombres de los objetos y lo que va a aparecer visualmente.

Cada opción del menú modificará el contenido del objeto correspondiente. El código debe ser como el siguiente:

```
private void saludoToolStripMenuItem_Click(object sender,
EventArgs e)
{
    // Se cambia el texto de la etiqueta
    lblTexto.Text = "Hola a todos";
}
```

4. Programación gráfica

```
private void despedidaToolStripMenuItem_Click(object sender,
    EventArgs e)
{
    // Se cambia el texto de la etiqueta
    lblTexto.Text = "Hasta la vista";
}

private void nombreDePilaToolStripMenuItem_Click(object sender,
    EventArgs e)
{
    // Se cambia el texto de la etiqueta
    lblTexto.Text = "Mi nombre es Borja";
}

private void verdeToolStripMenuItem_Click(object sender,
    EventArgs e)
{
    // Se cambia la imagen del semáforo
    picSemaforo.Image =
        Image.FromFile(@"..\img\SemaforoVerde.jpg");
}

private void ámbarToolStripMenuItem_Click(object sender,
    EventArgs e)
{
    // Se cambia la imagen del semáforo
    picSemaforo.Image =
        Image.FromFile(@"..\img\SemaforoAmbar.jpg");
}

private void rojoToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Se cambia la imagen del semáforo
    picSemaforo.Image =
        Image.FromFile(@"..\img\SemaforoRojo.jpg");
}

private void imagen1ToolStripMenuItem_Click(object sender,
    EventArgs e)
{
    // Se cambia la imagen de fondo de la ventana
    this.BackgroundImage =
        Image.FromFile(@"..\img\Fondo1.jpg");
}

private void imagen2ToolStripMenuItem_Click(object sender,
    EventArgs e)
```

```
// Se cambia la imagen de fondo de la ventana
this.BackgroundImage =
    Image.FromFile(@".\img\Fondo2.jpg");
}

private void salirToolStripMenuItem_Click(object sender,
    EventArgs e)
{
    // Se cierra la ventana
    this.Close();
}
```

En el código se puede apreciar que las propiedades para cambiar las imágenes son *Image* para las *PictureBox* y *BackgroundImage* para el fondo de la ventana. Las imágenes se cargan desde el disco duro, para lo cual hay que indicar la ruta en la que se encuentran (puede ser relativa, como en el código anterior, o absoluta). El hecho de utilizar la arroba (@) es para que se interprete el contenido.

En el siguiente ejemplo lo que se va a hacer es añadir un nuevo elemento a los que ya hay en el ejemplo anterior. En este caso, el nuevo elemento es una barra de herramientas.

Las barras de herramientas suelen tener las opciones más habituales de los menús. En este caso se van a poner las opciones que modifican la etiqueta, las que modifican la imagen y la de salir de la aplicación.

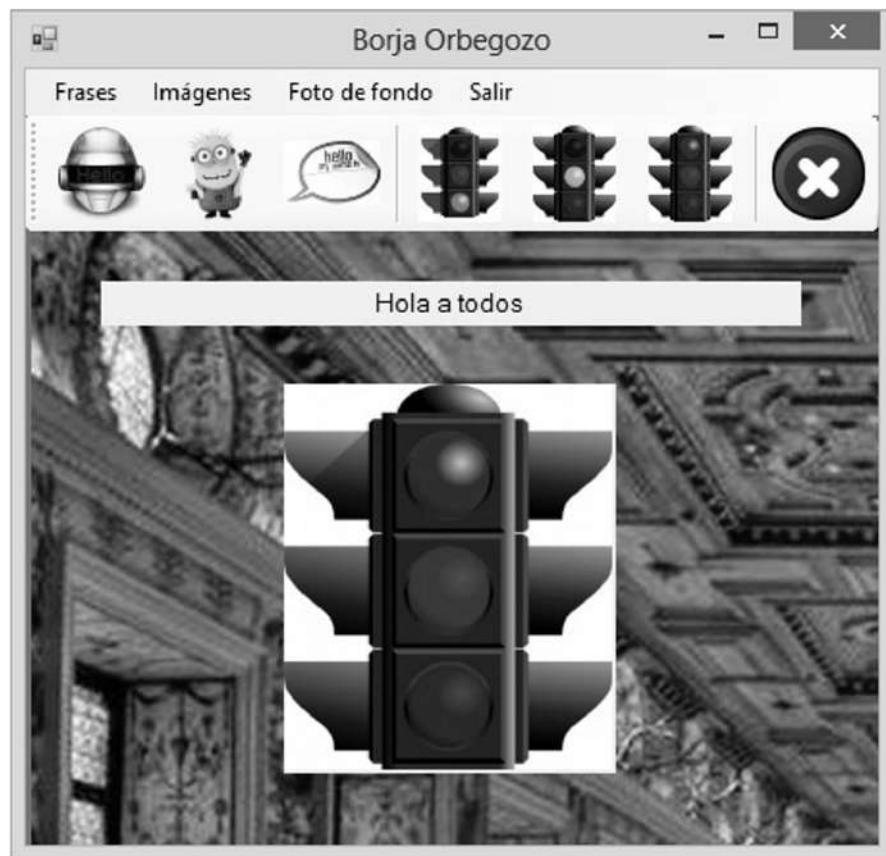
En las barras de herramientas lo que se colocan son botones que al pulsarlos ejecutarán las acciones asociadas a ellos. Además, para organizar los grupos de botones, se pueden utilizar separadores. Estos mismos separadores también pueden usarse en las opciones de los submenús.

Para este ejemplo, los tipos de herramientas que se van a utilizar son cuatro:

- *Label*: etiquetas que contienen texto fijo o variable desde código.
- *PictureBox*: controles de imagen.
- *MenuStrip*: menús principales.
- *ToolStrip*: barras de herramientas.

En el diseño se debe obtener una ventana como la de la imagen siguiente.

4. Programación gráfica



El código del menú va a ser el mismo que el del ejemplo anterior y lo que hay que hacer es añadir la parte de la barra de herramientas. Como se ha hecho en ejemplos anteriores, se va a reutilizar el código existente y únicamente se van a realizar llamadas a las funciones que ya hay en el menú.

El código que se debe añadir en la aplicación debe ser como el siguiente:

```
private void tbuSaludo_Click(object sender, EventArgs e)
{
    // Se llama a la función del menú Saludo
    saludoToolStripMenuItem_Click(sender, e);
}

private void tbuDespedida_Click(object sender, EventArgs e)
{
    // Se llama a la función del menú Despedida
```

```
}

private void tbuNombre_Click(object sender, EventArgs e)
{
    // Se llama a la función del menú Nombre de pila
    nombreDePilaToolStripMenuItem_Click(sender, e);
}

private void tbuVerde_Click(object sender, EventArgs e)
{
    // Se llama a la función del menú Verde
    verdeToolStripMenuItem_Click(sender, e);
}

private void tbuAmbar_Click(object sender, EventArgs e)
{
    // Se llama a la función del menú Ámbar
    ambarToolStripMenuItem_Click(sender, e);
}

private void tbuRojo_Click(object sender, EventArgs e)
{
    // Se llama a la función del menú Rojo
    rojoToolStripMenuItem_Click(sender, e);
}

private void tbuSalir_Click(object sender, EventArgs e)
{
    // Se llama a la función del menú Salir
    salirToolStripMenuItem_Click(sender, e);
}
```

En el siguiente ejemplo lo que se va a hacer es añadir un nuevo tipo de elemento a los ejemplos anteriores. En este caso se añaden los menús contextuales.

--- 4. Programación gráfica ---

Se va a poner un menú contextual a la etiqueta que permita modificar su contenido con las opciones que hay en el menú principal. Se va a poner otro menú contextual a la imagen que permita modificarla también de la misma manera que mediante el menú principal. Y, por último, se va a poner un menú contextual al fondo del propio formulario que permitirá modificar la imagen de fondo.

Para este ejemplo, los tipos de herramientas que se van a utilizar son cinco:

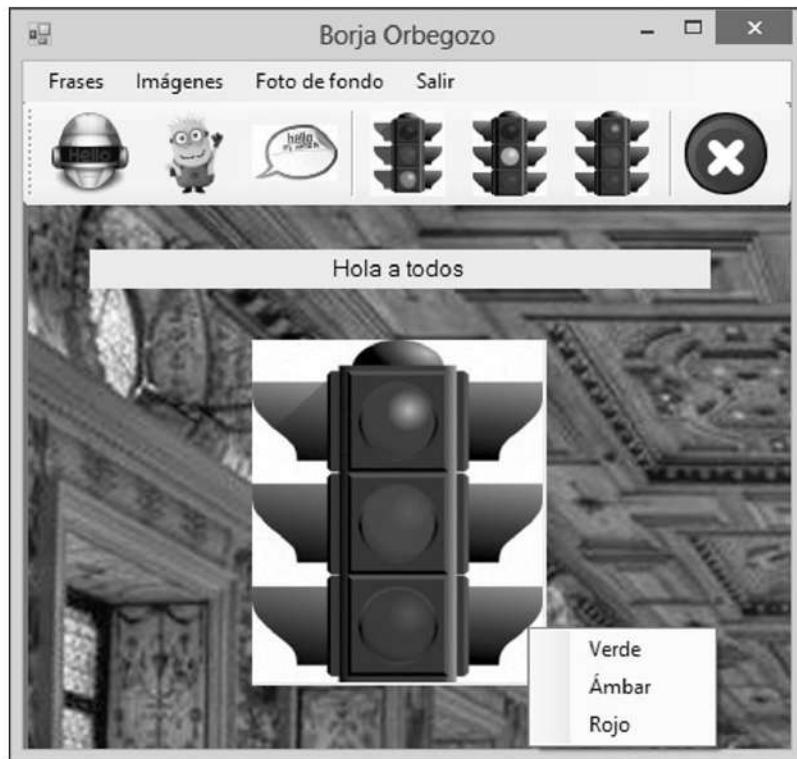
- *Label*: etiquetas que contienen texto fijo o variable desde código.
- *PictureBox*: controles de imagen.
- *MenuStrip*: menús principales.
- *ToolStrip*: barras de herramientas.
- *ContextMenuStrip*: menús contextuales.

Cuando se pulse en cada elemento, saldrá el menú contextual correspondiente. Para ello hay que establecer en tiempo de diseño la asociación entre los tres menús contextuales y las herramientas desde las que se van a mostrar.

El siguiente código es el que deberá aparecer tras asignar la propiedad *ContextMenuStrip* de cada herramienta al menú asociado.

```
// Menú contextual de frases para la etiqueta  
this.lblTexto.ContextMenuStrip = this.cmeFrases;  
  
// Menú contextual de imágenes para la imagen  
this.picSemaforo.ContextMenuStrip = this.cmeImagen;  
  
// Menú contextual de imágenes de fondo para el formulario  
this.ContextMenuStrip = this.cmeFondo;
```

Una vez realizadas las asignaciones, en tiempo de ejecución aparecerán los menús en función del lugar en el que se pulse con el botón derecho del ratón. La siguiente imagen corresponde a la pulsación sobre el elemento de imagen de la ventana.



El código del menú y el de la barra de herramientas van a ser los mismos que en el ejemplo anterior y lo que hay que hacer es añadir la parte de los menús contextuales. Como se ha hecho en ejemplos anteriores, se va a reaprovechar el código existente y únicamente se van a realizar llamadas a las funciones que ya hay en el menú.

```
private void saludoToolStripMenuItem1_Click(object sender,
EventArgs e)
{
    // Se llama a la función del menú Saludo
    saludoToolStripMenuItem_Click(sender, e);
}

private void despedidaToolStripMenuItem1_Click(object sender,
EventArgs e)
{
    // Se llama a la función del menú Despedida
    despedidaToolStripMenuItem_Click(sender, e);
}
```

4. Programación gráfica

```
private void nombreDePilaToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Se llama a la función del menú Nombre de pila
    nombreDePilaToolStripMenuItem_Click(sender, e);
}

private void verdeToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Se llama a la función del menú Verde
    verdeToolStripMenuItem_Click(sender, e);
}

private void ámbarToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Se llama a la función del menú Ámbar
    ambarToolStripMenuItem_Click(sender, e);
}

private void rojoToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Se llama a la función del menú Rojo
    rojoToolStripMenuItem_Click(sender, e);
}

private void imagen1ToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Se llama a la función de la imagen 1
    imagen1ToolStripMenuItem_Click(sender, e);
}

private void imagen2ToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Se llama a la función de la imagen 2
    imagen2ToolStripMenuItem_Click(sender, e);
}
```

Cuadros de diálogo

La lista de los cuadros de diálogo que tenemos a nuestra disposición en el cuadro de herramientas es la que se encuentra en la imagen siguiente.



Los cuadros de diálogo permiten abrir ventanas específicas para determinados elementos, como por ejemplo, la selección del tipo de letra o de los colores. Son ventanas estándares que se pueden lanzar desde *Visual C#* controlando sus contenidos y sus selecciones.

Permiten también realizar acciones como la búsqueda de archivos y el guardado de los mismos en el disco.

También existen otros cuadros de diálogo específicos que se muestran directamente desde el código. Sirven para mostrar mensajes de texto con diferentes opciones y también para hacer preguntas con respuestas concretas al usuario de la aplicación.

La mejor forma de ver el funcionamiento y el aspecto de los diferentes diálogos, es mediante un ejemplo. Para ello se va a crear una ventana en la que haya cuatro botones diferentes que realicen las siguientes acciones:

- Buscar archivo.
- Guardar archivo.
- Cambiar fuente.
- Cambiar color.

Además de eso, en la ventana habrá una caja de texto en la que se solicitará el nombre del archivo que se quiere buscar o que se quiere guardar.

Este ejemplo va a tener varios objetivos: aprender el uso del cuadro de diálogo de búsqueda, aprender el uso del cuadro de diálogo para guardar un archivo, aprender el uso del cuadro de diálogo para seleccionar una fuente y aprender el uso del cuadro de diálogo para elegir un color.

4. Programación gráfica

Para este ejemplo, los tipos de herramientas que se van a utilizar son seis:

- *Label*: etiquetas que contienen texto fijo o variable desde código.
- *TextBox*: cajas de texto.
- *OpenFileDialog*: cuadros de diálogo para abrir archivos.
- *SaveFileDialog*: cuadros de diálogo para guardar archivos.
- *FontDialog*: cuadros de diálogo para elegir fuentes.
- *ColorDialog*: cuadros de diálogo para elegir colores.

En el diseño se debe obtener una ventana como la de la siguiente imagen.

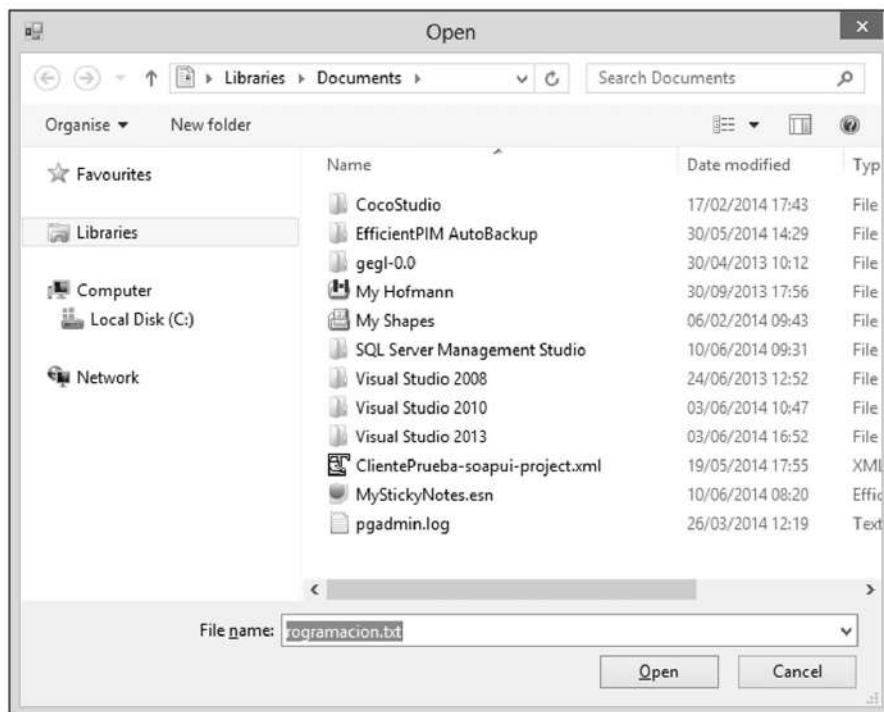


Lo mejor para explicar este ejemplo es ver el código que debe ir asociado a cada uno de los botones. Empezando por el botón *Buscar archivo*, como

código interno es el siguiente:

```
private void butBuscar_Click(object sender, EventArgs e)
{
    // Se asigna el nombre del archivo que hay que buscar
    opeDialog.FileName = txtNombre.Text;
    // Se abre el cuadro de diálogo
    opeDialog.ShowDialog();
}
```

Como se puede apreciar, en la propiedad *File name* aparece el nombre del archivo que se ha introducido en la caja de texto. El cuadro de diálogo que se va a abrir es el de esta imagen.



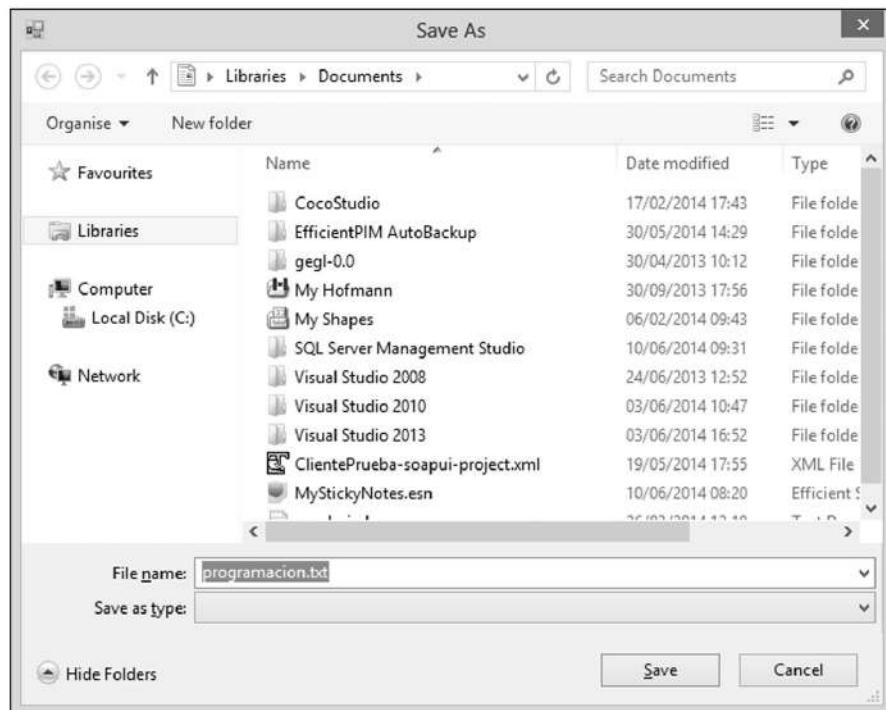
Es el cuadro de diálogo estándar en el que se puede elegir la carpeta en la que se va a buscar el archivo. El nombre incluido es el que aparecerá por defecto, aunque se podría modificar en el propio cuadro de diálogo.

El botón *Abrir archivo* debe tener el siguiente código:

4. Programación gráfica

```
private void butGuardar_Click(object sender, EventArgs e)
{
    // Se asigna el nombre del archivo que hay que buscar
    savDialog.FileName = txtNombre.Text;
    // Se abre el cuadro de diálogo
    savDialog.ShowDialog();
}
```

Como se aprecia, el código es muy similar al anterior, pero abre el cuadro de diálogo correspondiente a guardar archivos. La ventana que se abre es la de la siguiente imagen.



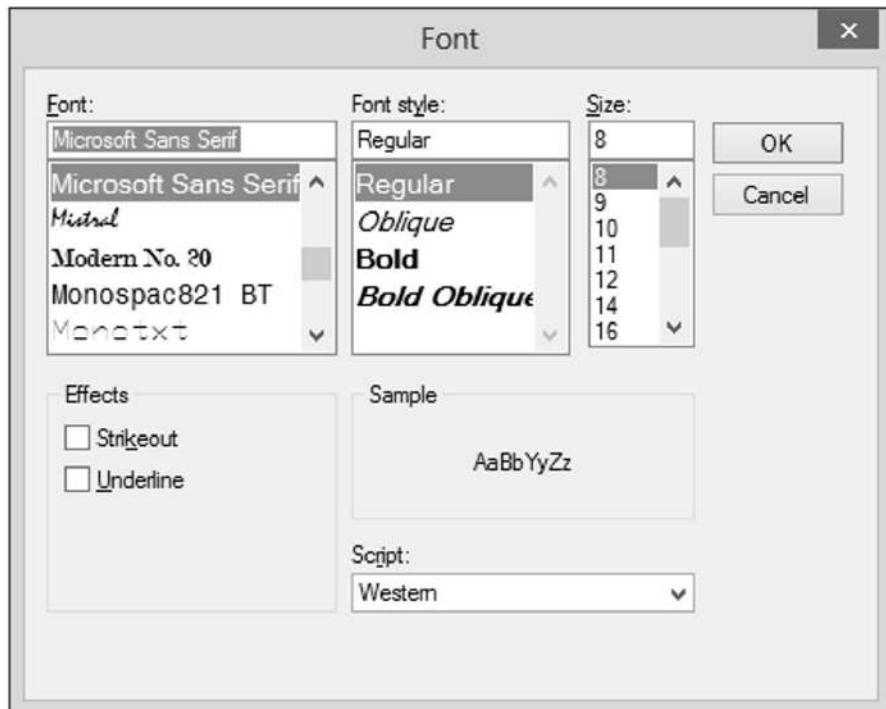
El funcionamiento es equivalente al del anterior, pero en este caso sirve para guardar en lugar de para buscar.

Para el botón *Cambiar fuente* se debe incluir el siguiente código:

```
private void butFuentes_Click(object sender, EventArgs e)
```

```
{  
    // Se abre el cuadro de diálogo  
    fonDialogo.ShowDialog();  
}
```

En este caso no hay que hacer nada específico más que la llamada al cuadro de diálogo de selección de fuente, que tendrá el aspecto de esta ventana:



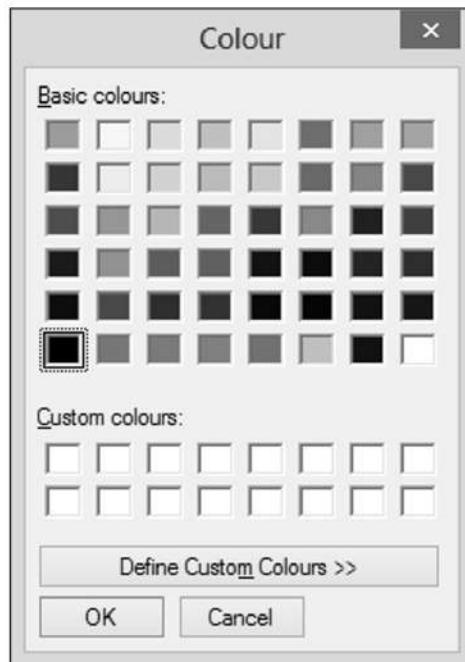
En este cuadro de diálogo se pueden elegir la fuente, el estilo, el tamaño y también las características especiales de la misma, como tachado y subrayado.

El último botón, *Cambiar color*, debe incluir el siguiente código:

4. Programación gráfica

```
private void butColores_Click(object sender, EventArgs e)
{
    // Se abre el cuadro de diálogo
    colDialogo.ShowDialog();
}
```

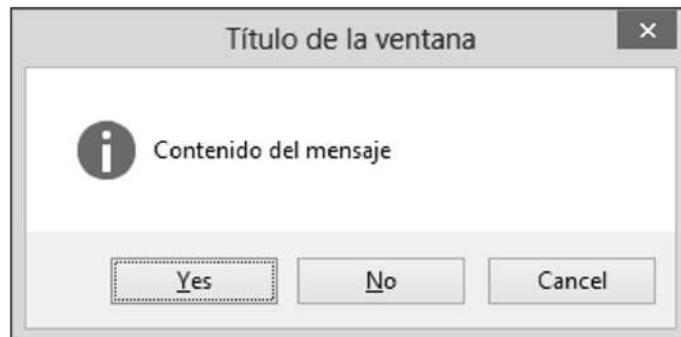
La llamada al cuadro de diálogo de selección del color abrirá la siguiente ventana:



En este cuadro de diálogo se puede elegir el color entre las opciones básicas o definirlo de manera personalizada usando la mezcla de colores.

Continuando con el mismo ejemplo, se va a hacer que cuando arranque la aplicación, salga de entrada un cuadro de diálogo en el que se pregunte si se quiere iniciar la aplicación o no.

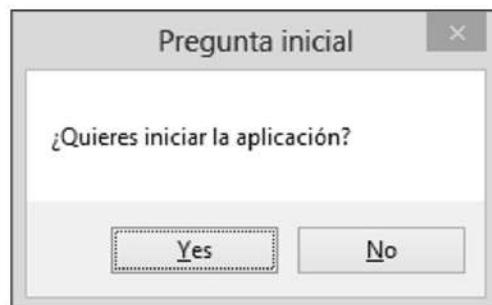
El aspecto que tienen estos cuadros de diálogo es como el que se muestra en la siguiente imagen.



En la imagen se pueden apreciar cuatro tipos diferentes de elementos personalizables:

- El título de la ventana.
- El ícono que se muestra en la ventana.
- El contenido del mensaje que aparece en la ventana.
- Los botones que permiten las diferentes opciones para elegir.

En este caso, lo que se quiere es una pregunta simple para saber si se quiere iniciar la aplicación y que permita responder *sí* o *no*. Como en esta imagen:



4. Programación gráfica

Para que se muestre así, el código que hay que escribir debe ser como el que se presenta seguidamente.

```
private void frmEjercicio12_Load(object sender, EventArgs e)
{
    // Se inicializan los diferentes elementos
    string sMensaje = "¿Quieres iniciar la aplicación?";
    string sTitulo = "Pregunta inicial";
    MessageBoxButtons mBotones = MessageBoxButtons.YesNo;
    DialogResult sResultado;
    // Se muestra el cuadro de diálogo
    sResultado = MessageBox.Show(sMensaje, sTitulo, mButtons);
    // Se chequea la pulsación del botón
    if (sResultado == System.Windows.Forms.DialogResult.No)
    {
        // Se cierra la aplicación
        this.Close();
    }
}
```

En este código se utiliza el comando *MessageBox.Show*, al que se le están pasando como parámetros el mensaje de la ventana, el título y los botones que se quieren usar. Los botones aparecerán en el idioma en el que se tenga Windows instalado, por ese motivo en las imágenes aparecen en inglés.

Las opciones que se han incluido en *MessageBoxButtons* son las de *sí* o *no*, pero pueden utilizarse todas estas posibilidades:

Opción	Descripción
OK	botón de <i>Aceptar</i>
OKCancel	botones de <i>Aceptar</i> y <i>Cancelar</i>
YesNo	botones de <i>Sí</i> y <i>No</i>
YesNoCancel	botones de <i>Sí</i> , <i>No</i> y <i>Cancelar</i>
RetryCancel	botones de <i>Reintentar</i> y <i>Cancelar</i>
AbortRetryIgnore	botones de <i>Abortar</i> , <i>Reintentar</i> e <i>Ignorar</i>

En función del botón que se pulse, las respuestas que se pueden recibir son las de la siguiente lista:

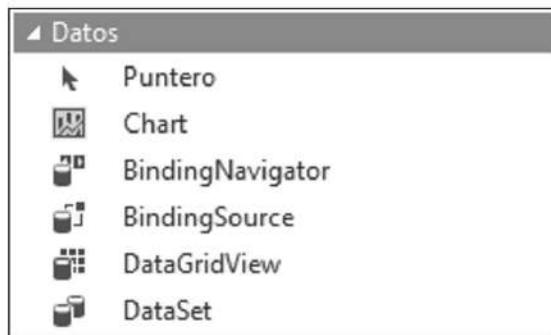
Botón	Opción
<i>Abort</i>	<i>DialogResult.Abort</i>
<i>Cancel</i>	<i>DialogResult.Cancel</i>
<i>Ignore</i>	<i>DialogResult.Ignore</i>
<i>No</i>	<i>DialogResult.No</i>
<i>OK</i>	<i>DialogResult.OK</i>
<i>Retry</i>	<i>DialogResult.Retry</i>
<i>Yes</i>	<i>DialogResult.Yes</i>

Por defecto no se pone ningún ícono en la ventana de mensajes, pero las posibilidades que se tienen son las siguientes:

Icono	Opción	Descripción
	<i>None</i>	Ninguno
	<i>Asterisk, Information</i>	Información
	<i>Hand, Stop, Error</i>	Aviso de error
	<i>Question</i>	Pregunta
	<i>Exclamation, Warning</i>	Aviso

Acceso a bases de datos

La lista de herramientas de acceso a bases de datos que tenemos a nuestra disposición en el cuadro de herramientas es la que se encuentra en la imagen siguiente:



El elemento principal de las herramientas de acceso a bases de datos es el enlace con el navegador. La herramienta *BindingNavigator* permite moverse por los registros de la tabla de una base de datos de manera directa.

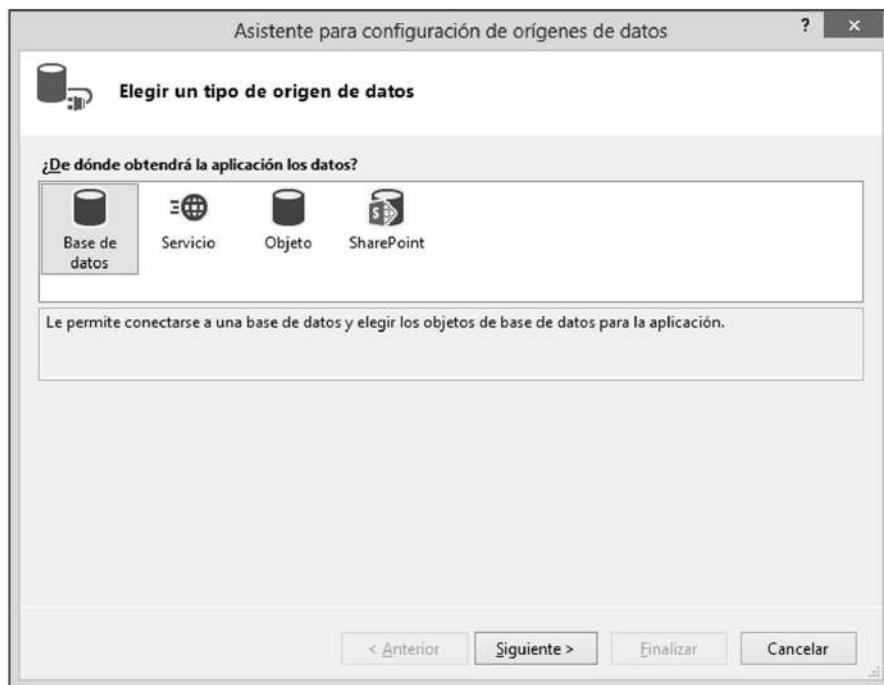
Para enlazar el navegador con la tabla, se utiliza el *BindingSource*, que permite definir la fuente del enlace.

Aunque lo habitual cuando se realiza el enlace es elegir una base de datos, hay varias opciones más que se detallan a continuación:

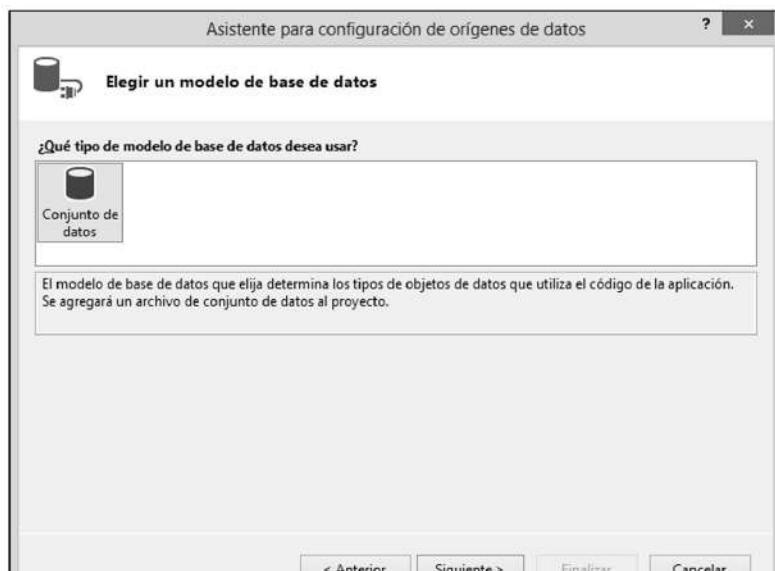
- *Base de datos*: permite que la aplicación se conecte y modifique los datos incluidos en una base de datos.
- *Servicio*: permite que la aplicación se conecte y funcione con los datos y métodos de un servicio de *Windows Communication Foundation*, de un servicio de datos o de un servicio web.
- *Objeto*: permite que la aplicación funcione con los datos incluidos en objetos de la propia aplicación.
- *Sharepoint*: permite que la aplicación funcione con datos desde un sitio de *Sharepoint*.

Para realizar el enlace hay que lanzar el *Asistente para configuración de orígenes de datos*. La primera ventana que sale cuando se lanza el asistente es la siguiente:

Desarrollo de aplicaciones C# con Visual Studio .NET

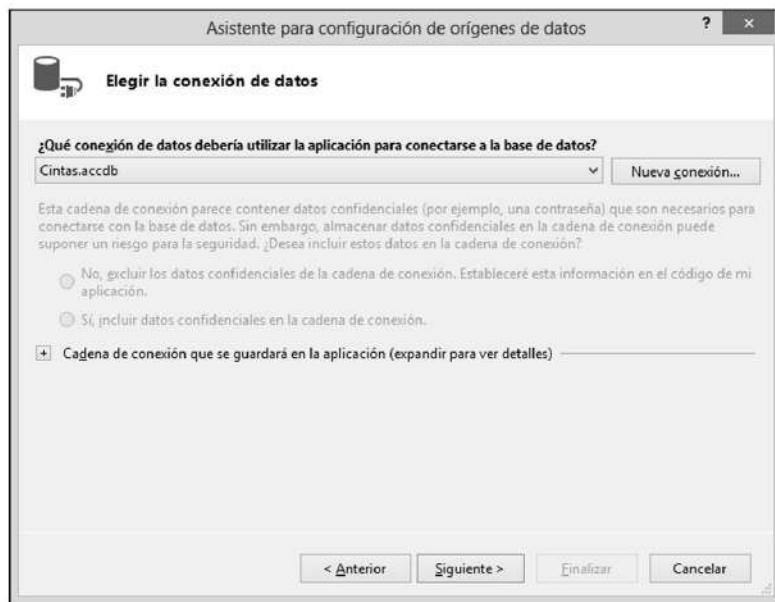


En esta primera ventana se va a elegir uno de los tipos de origen de datos explicados antes. Una vez que se ha seleccionado uno (en este caso se va a optar por una base de datos, que es lo más estándar), se pulsa en el botón *Siguiente* para pasar a la siguiente ventana.



4. Programación gráfica

En la nueva ventana hay que elegir el modelo de base de datos. En este caso la única opción es la de conjuntos de datos, así que se debe marcar dicha opción y pulsar en el botón *Siguiente* para continuar.



El siguiente punto permite elegir la conexión de datos que se va a utilizar. Para ello hay que pulsar en el botón *Nueva conexión* y desde ahí elegir entre las opciones que se tienen en el propio *Windows*.

Una vez que se tenga la nueva conexión, se puede ver la cadena de conexión generada expandiendo la opción *Cadena de conexión que se guardará en la aplicación*.

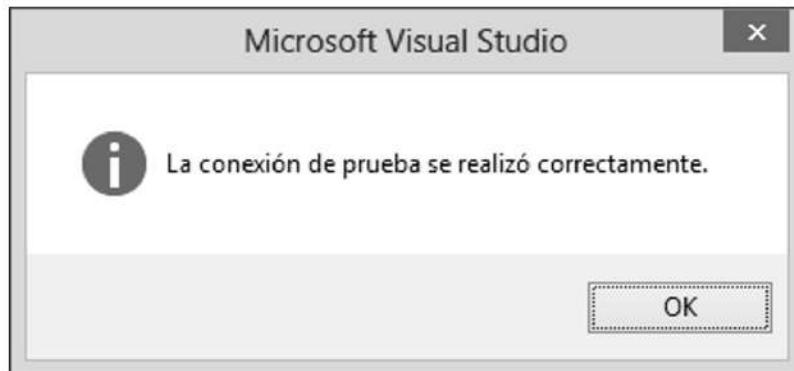
En este caso, se va a elegir una base de datos de *Microsoft Access*, cuya utilización e implementación es muy sencilla. Tras elegir el tipo de base de datos, se elegirá el archivo correspondiente a la base de datos que se vaya a utilizar. Esto se puede ver en la siguiente imagen.



En función de la base de datos, será necesario o no introducir un nombre de usuario y una contraseña.

Cuando ya se haya elegido el archivo y se hayan introducido el usuario y la contraseña si procede, es conveniente pulsar el botón *Probar conexión* para verificar que todo sea correcto.

Si la prueba de conexión ha sido correcta, debe aparecer una ventana como la siguiente:



4. Programación gráfica

Después de terminar la prueba y volver a la ventana anterior, se visualizará un cuadro de diálogo en el que se preguntará qué hacer con la conexión que se ha generado. El usuario debe decidir si el archivo de datos se va a incorporar al proyecto que se está desarrollando o se va a dejar solamente de forma externa.



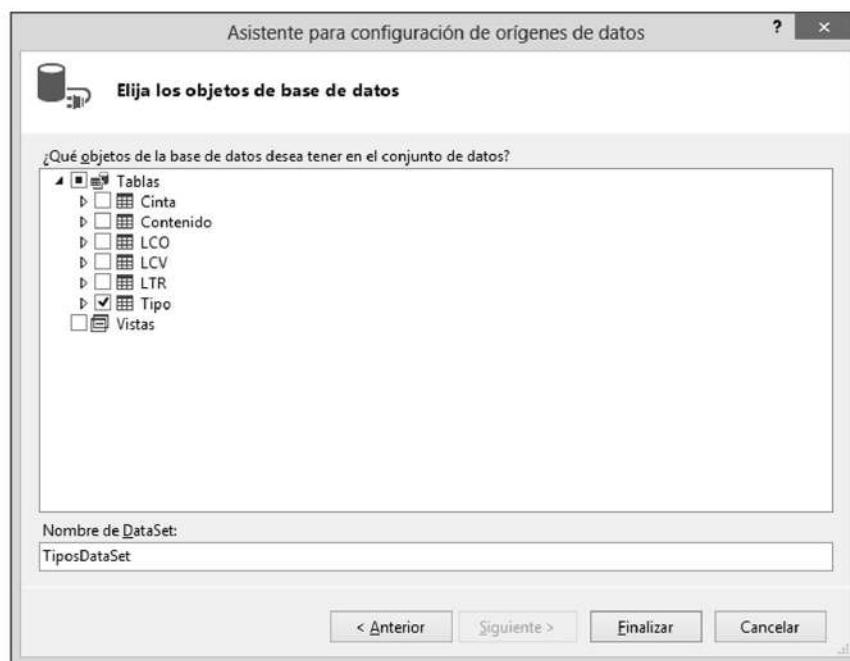
Cuando ya se haya decidido la opción preferida, se puede continuar el proceso.

En la siguiente ventana se pregunta si se desea guardar o no la conexión que se ha creado. Guardar la conexión tiene sus ventajas si se va a utilizar en más ocasiones. Si es así, entonces se le puede poner un nombre y más adelante se puede volver a usar. La ventana que aparece es la siguiente:



Una vez que se haya elegido el nombre de la conexión o se haya optado por no guardarla, se pulsa el botón *Siguiente* para continuar.

El siguiente paso es la selección de los objetos que se van a utilizar. Aparecerán desplegadas las tablas de la base de datos y los campos de dichas tablas. La ventana de selección es la siguiente:



Lo más habitual es elegir una de las tablas con todos sus campos, pero eso depende de lo que se esté desarrollando.

Para ver el funcionamiento de todo esto lo mejor es utilizar un ejemplo. Para ello se va a realizar un programa en el que se enlacen unas cajas de texto con una base de datos.

Se va a elegir una tabla que tenga un par de campos y se va a permitir el recorrido por los registros de la tabla desde la ventana de la aplicación, usando para ello una barra de navegación.

También debe haber un botón *Salir* para finalizar el programa.

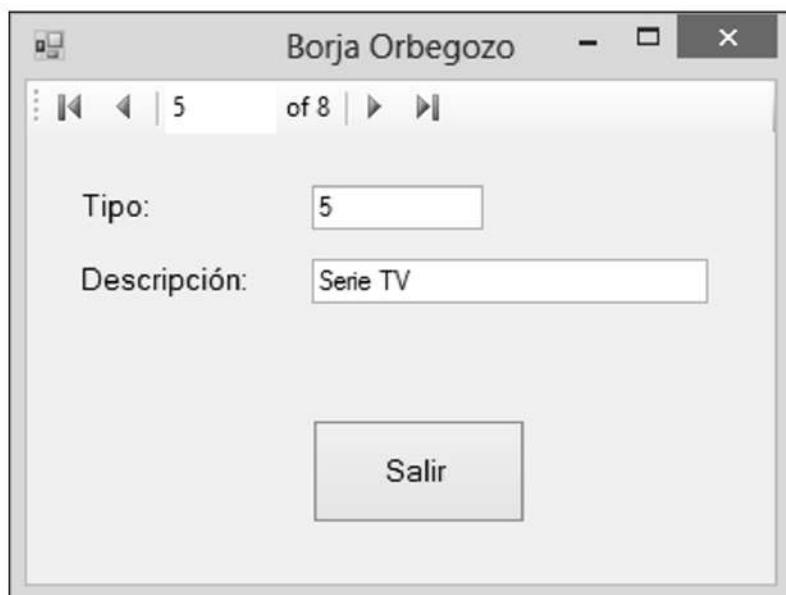
El objetivo de este ejemplo es aprender a enlazar una tabla con los elementos de pantalla.

4. Programación gráfica

Para este ejemplo, los tipos de herramientas que se van a utilizar son cinco:

- *Label*: etiquetas que contienen texto fijo o variable desde código.
- *TextBox*: cajas de texto.
- *Button*: botones de comando.
- *BindingNavigator*: barras de navegación.
- *BindingSource*: enlaces a fuentes de datos.

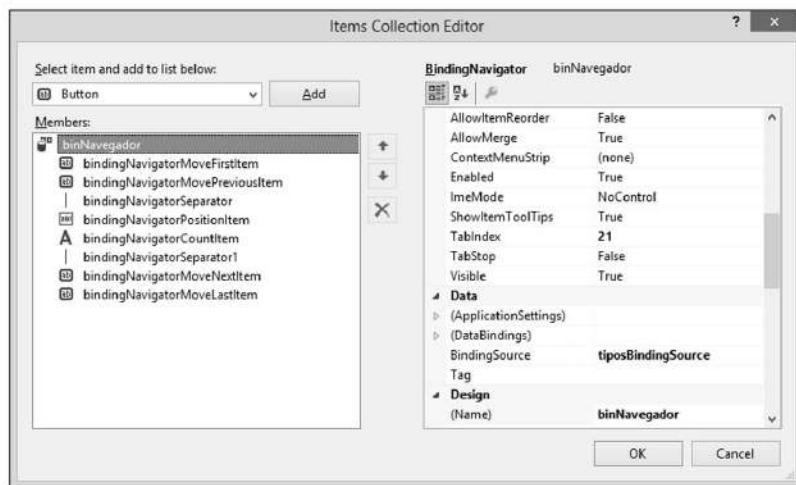
En el diseño se debe obtener una ventana como la de la imagen siguiente.



Este ejemplo no lleva código de programación, excepto la parte de salida del programa. Todo lo que hay que hacer es crear primero el enlace tal y como se ha explicado anteriormente y después enlazar lo que se ha creado con las cajas de texto dispuestas en la ventana del programa.

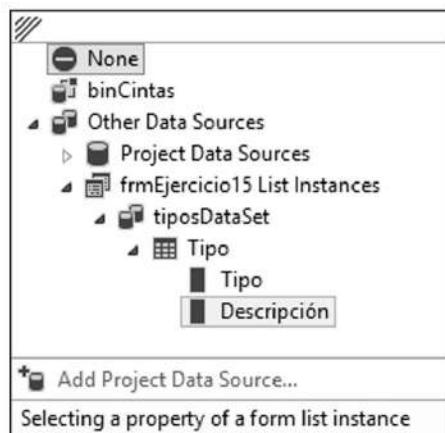
De entrada, no está de más comentar que se pueden elegir los botones que se van a visualizar en la barra de navegación. Por defecto aparecen todos los que hay, pero esto se puede editar y modificar. En el ejemplo anterior se han eliminado los botones para añadir elementos y para borrar elementos, puesto que lo que se busca en este caso sólo es la navegación.

En la propiedad *Items* de la barra de navegación se eligen los botones que van a verse en la barra. Esta colección se puede editar. Para este ejemplo se ha dejado la siguiente colección:



Por lo demás, la propiedad más importante de la barra de navegación para este caso es la propiedad *BindingSource*, puesto que es en dicha propiedad donde se indica el origen de los datos que se van a utilizar.

En el caso de las cajas de texto, la propiedad que hay que modificar es la propiedad *Text* dentro de *DataBindings*. En dicha propiedad no solamente se va a elegir el origen de los datos, sino más en concreto, el campo que se va a asociar con el control. Por ejemplo, para la caja de texto *txtDescripcion*, lo que se debe colocar es *tiposBindingSource – Descripción*, que aparece al seleccionar lo que se ve en la siguiente imagen.



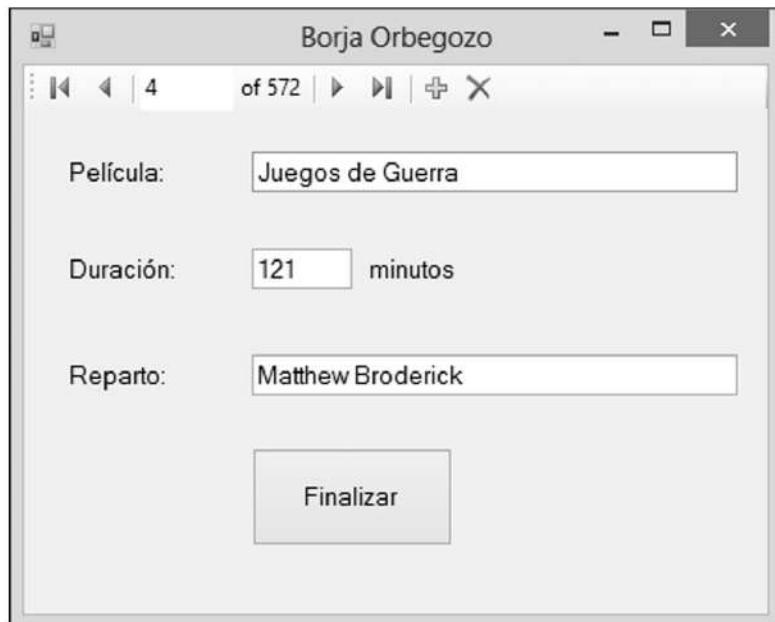
4. Programación gráfica

En el siguiente ejemplo, se va a hacer prácticamente lo mismo, pero permitiendo que además de la navegación, se puedan añadir y eliminar elementos.

Se va a elegir otra tabla, se va a asociar dicha tabla a una barra de navegación completa y se van a permitir todas las acciones posibles.

El objetivo de este ejemplo es el mismo que el del ejemplo anterior, sumándole la parte de añadir y eliminar elementos dentro de la navegación.

Los tipos de herramientas que se van a utilizar para este ejemplo son los mismos que los del ejemplo anterior y el aspecto de la ventana que se va a visualizar en ejecución es el de la siguiente imagen.



De la misma forma que en el anterior, este ejemplo se genera a base de diseño en lugar de hacerlo por código. En este caso se va a enlazar con una tabla diferente que tendrá los campos que se ven en la ventana y cada caja de texto va a estar asociada a su campo correspondiente.

En este caso se ha mantenido la barra de navegación por defecto, con lo que se ve que aparte de los botones para moverse al primero, anterior, siguiente y último registro, están los botones para añadir nuevos elementos y para eliminar el elemento que se esté viendo en ese momento.

Para el siguiente ejemplo se va a utilizar el acceso a bases de datos, pero esta vez por código en lugar de usar las herramientas, puesto que son bastante limitadas a la hora de hacer muchas cosas.

En este nuevo ejemplo se va a utilizar una base de datos que tenga una tabla muy simple de películas en la que se incluyan los campos:

- *pelCodigo*: código asignado a la película.
- *pelTitulo*: título en castellano.
- *pelDuracion*: duración en minutos.

En la ventana se va a tener una caja de texto para cada campo con su correspondiente etiqueta asociada.

Para realizar la gestión de los datos habrá cuatro botones principales:

- *Consulta*: al dar un código, se obtiene el título y la duración de la película.
- *Alta*: crea un nuevo registro con los datos de las cajas de texto.
- *Baja*: elimina el registro que se esté viendo en la ventana usando el código como clave de eliminación.
- *Modificación*: modifica el registro del código de la película que se ve en la ventana y le pone el título y la duración que se indiquen.

Además de los botones principales, cada acción se deberá refrendar con dos botones auxiliares:

- *Aceptar*: se acepta la acción realizada y se ejecutan los cambios.
- *Cancelar*: no se hace nada y se cancela la acción solicitada.

Los botones auxiliares deberán estar desactivados hasta que se haya elegido alguna de las acciones principales. En el momento en que se elija una acción, se activarán los botones auxiliares y se desactivarán los botones con las acciones principales. Es decir, que los principales y auxiliares nunca estarán activos a la vez.

También habrá un botón para finalizar el programa.

4. Programación gráfica

Los objetivos de este ejemplo van a ser, por un lado, aprender a realizar la gestión de los datos de una tabla procedente de una base de datos desde código; y por el otro, ver cuáles son las instrucciones *SQL* estándares para realizar consultas, inserciones, eliminaciones y modificaciones.

Para este ejemplo, los tipos de herramientas que se van a utilizar son tres:

- *Label*: etiquetas que contienen texto fijo o variable desde código.
- *TextBox*: cajas de texto.
- *Button*: botones de comando.

El aspecto que va a tener la ventana cuando se ejecute la aplicación por primera vez va a ser como el de la siguiente imagen.



En este ejemplo se va a utilizar una base de datos de *Microsoft Access* en la que haya una tabla llamada *Películas* con tres campos que van a corresponder a las cajas de texto creadas en la ventana. El aspecto del diseño de la tabla debe ser como el de la siguiente imagen.

Películas		
	Field Name	Data Type
!	Código	AutoNumber
	Título	Text
	Duración	Number

En la imagen se puede apreciar que el campo *Código* es la clave de la tabla y además es de tipo autonumérico. Ello quiere decir que cuando se hagan las inserciones de registros, dicho campo tomará el siguiente valor libre y no será necesario especificarlo.

Antes de comenzar a explicar el código, hay que conocer las instrucciones básicas de *SQL (Structured Query Language)* que nos permitirán realizar todas las operaciones con los datos.

La instrucción de inserción básica tiene la siguiente estructura:

```
INSERT INTO tabla [(campo1[, campo2[, ...]])]
VALUES (valor1[, valor2[, ...]])
```

En esta instrucción lo que se hace es insertar un registro en la tabla indicada, especificando los campos en los que se van a introducir los datos (no es obligatorio ponerlos si se van a usar todos) y los valores para cada uno de los campos en el mismo orden en que están descritos. Los valores que sean de tipo texto deben ir entrecomillados con comillas simples.

Existe otro formato para esta misma instrucción:

```
INSERT INTO tabla [(campo1[, campo2[, ...]])]
SELECT [tabla.]campo1[, campo2[, ...]] FROM tabla(s)
```

En este formato se pueden realizar varias inserciones a la vez. Dependerá del número de registros que devolverá la instrucción de selección que se va a comentar más adelante.

4. Programación gráfica

La instrucción de modificación tiene la estructura siguiente:

```
UPDATE table  
SET campo=valor  
WHERE condición
```

En este caso se modifica el valor de un campo de una tabla en función de la condición que se haya puesto. Hay que tener en cuenta que si varios registros cumplen dicha condición, entonces se modificará el campo indicado en todos esos registros. Si no se pone ninguna condición, entonces la modificación se hará en todos los registros de la tabla.

La instrucción de eliminación tiene esta estructura:

```
DELETE FROM table  
WHERE condición
```

Esta instrucción elimina de la tabla todos los registros que cumplen la condición que se haya descrito. También en este caso, si la condición la cumplen varios registros, se eliminan todos ellos. Si no se pone ninguna condición, se eliminan todos los registros de la tabla.

Por último, la instrucción de selección tiene este formato:

```
SELECT {*} / tabla.* / [tabla.]campo1 [AS alias1]  
[, [tabla.]campo2 [AS alias2] [, ...]]}  
FROM tabla(s)  
[WHERE condición]  
[GROUP BY campo(s)]  
[HAVING condición]  
[ORDER BY campo(s) [ASC / DESC]]
```

En esta instrucción hay varias cláusulas que conviene describir. En la cláusula *SELECT* se indican los campos que se quieren obtener. Si se pone un asterisco, se indica que se quieren devolver todos los campos de todas las tablas que intervengan en la consulta. Si se pone un nombre de tabla seguido de un asterisco, se devuelven todos los campos de la tabla indicada. Si se ponen campos sueltos, se devolverán dichos campos. Cada uno de los campos puede llevar un alias o sobrenombre. También se pueden utilizar funciones especiales para campos numéricos como *SUM* (suma de contenidos), *AVG* (valor medio), *MIN* (valor mínimo) y *MAX* (valor máximo).

En la cláusula *FROM* se indican las tablas que van a intervenir en cualquier parte de la instrucción.

En la cláusula *WHERE* se colocan las condiciones de selección. Si se ponen varias condiciones, se pueden utilizar los conectores *AND* (se deben evaluar a cierto las condiciones de cada lado) y *OR* (basta con que se evalúe a cierto una de las condiciones de los lados).

En la cláusula *GROUP BY* se pueden indicar grupos de campos para el cálculo de las cuatro funciones especiales indicadas en la primera cláusula.

La cláusula *HAVING BY* sirve para establecer condiciones que deben cumplir los elementos de la cláusula *GROUP BY*.

En la cláusula *ORDER BY* se pueden establecer los campos por los que se quieren ordenar los resultados. Se puede indicar además para cada campo si el orden se desea ascendente (*ASC*), es el valor por defecto, o descendente (*DESC*), hay que indicarlo explícitamente.

Respecto al código del ejemplo, dado que es bastante largo y hay bastantes elementos, lo mejor será ir paso a paso explicando cada una de sus partes.

Lo primero que se va a hacer es la declaración de las variables globales de la aplicación. Son variables que se van a utilizar en varios de los métodos y funciones. Su declaración es la siguiente:

```
// Variables principales
System.Data.OleDb.OleDbConnection conBD = new
    System.Data.OleDb.OleDbConnection();
String sAccion = " ";
```

La primera de ellas se va a utilizar para guardar la conexión con la base de datos y la segunda para saber qué acción se debe ejecutar.

Es importante declarar dos funciones que se puedan utilizar en cualquier aplicación que trate con bases de datos de este tipo. Se trata de una función para abrir una conexión con la base de datos y otra para cerrar dicha conexión. Es preferible tenerlas como funciones para poder usarlas desde varios lugares llamándolas sin más.

4. Programación gráfica

La función para abrir la conexión con la base de datos será como la siguiente:

```
public void abrirConexion()
{
    // Se crea el string de conexión a la base de datos
    conBD.ConnectionString = @"Provider=Microsoft.ACE.
        OLEDB.12.0; Data Source=.\\BD\\Grabaciones.accdb; Per-
        sist Security Info=False";
    try
    {
        // Se abre la base de datos
        conBD.Open();
    }
    catch (Exception ex)
    {
        MessageBox.Show("Error de conexión" + ex);
    }
}
```

Lo principal es establecer correctamente la cadena de conexión con la base de datos en cuestión. Si se quisiera cambiar el sistema gestor de base de datos, habría que modificar esta cadena de conexión, pero el resto permanecería invariable.

Es interesante también usar la instrucción *try... catch*, que permite realizar un control de las posibles excepciones que salten en el programa. Cuando salte una excepción al intentar abrir la conexión con la base de datos, se visualizará un cuadro de diálogo con un mensaje de error.

Para el cierre de la conexión se debe usar una función como la siguiente:

```
public void cerrarConexion()
{
    // Se chequea si está la conexión abierta
    if (conBD.State == ConnectionState.Open)
    {
        // Se cierra la conexión
        conBD.Close();
    }
}
```

En este caso se hace un chequeo para controlar si la conexión está abierta antes de cerrarla. De esta forma, si por un error se llamaría dos veces desde el código al cierre de la conexión, la segunda vez no daría problemas.

Teniendo en cuenta que los botones de las acciones van a estar activados cuando los de *Aceptar* y *Cancelar* estén desactivados y viceversa, es conveniente realizar dos funciones que realicen las instrucciones correspondientes para activar y desactivar cada fila de botones.

La función para desactivar los botones de acciones y activar los otros dos es la siguiente:

```
private void desactivarAcciones()
{
    butConsulta.Enabled = false;
    butAlta.Enabled = false;
    butBaja.Enabled = false;
    butModificacion.Enabled = false;
    butAceptar.Enabled = true;
    butCancelar.Enabled = true;
}
```

Y la función para activar los botones de acciones y desactivar los otros dos es la siguiente:

```
private void activarAcciones()
{
    butConsulta.Enabled = true;
    butAlta.Enabled = true;
    butBaja.Enabled = true;
    butModificacion.Enabled = true;
    butAceptar.Enabled = false;
    butCancelar.Enabled = false;
    // Se cancela la acción a realizar
    sAcción = "";
}
```

En esta segunda función también se vacía la variable de la acción que estuviera seleccionada, ya que para la siguiente vez el usuario deberá pulsar de nuevo alguno de los botones de acción.

Cuando arranca la aplicación, se ejecuta el evento *Load*, que va a abrir la conexión con la base de datos. Dicha conexión permanecerá abierta durante toda la ejecución del programa.

4. Programación gráfica

```
private void frmEjercicio17_Load(object sender, EventArgs e)
{
    // Se abre la conexión con la base de datos
    abrirConexion();
}
```

Como en los otros ejemplos, el botón para finalizar la aplicación debe tener el código de cierre de la ventana, pero en este caso también se deberá cerrar la conexión a la base de datos.

```
private void butSalir_Click(object sender, EventArgs e)
{
    // Se cierra la conexión antes de terminar
    cerrarConexion();
    // Se finaliza el programa
    this.Close();
}
```

Lo único que deben hacer los botones de las diferentes acciones es marcar la acción que se ha pulsado y también llamar a la función que desactiva y activa los botones correspondientes. El código para los cuatro botones es muy similar, tal y como se puede apreciar en las siguientes funciones:

```
private void butConsulta_Click(object sender, EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAcción = "Consulta";
    // Se activan y desactivan los botones correspondientes
    desactivarAcciones();
}

private void butAlta_Click(object sender, EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAcción = "Alta";
    // Se activan y desactivan los botones correspondientes
    desactivarAcciones();
}

private void butBaja_Click(object sender, EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAcción = "Baja";
    // Se activan y desactivan los botones correspondientes
    desactivarAcciones();
}
```

```
        }
        private void butModificacion_Click(object sender, EventArgs e)
        {
            // Se indica la acción que hay que realizar
            sAcción = "Modificación";
            // Se activan y desactivan los botones correspondientes
            desactivarAcciones();
        }
```

De los dos botones que quedan, el de *Cancelar* tiene un código muy sencillo, puesto que lo único que tiene que hacer es activar y desactivar los botones correspondientes y no hacer nada más. Debe también vaciar la variable de la acción, pero eso ya se hace en la misma función de activación de acciones.

```
        private void butCancelar_Click(object sender, EventArgs e)
        {
            // Se activan y desactivan los botones correspondientes
            activarAcciones();
        }
```

Sin embargo, el botón *Aceptar* es el que va a tener el grueso del código, puesto que es ahí donde se van a realizar todas las acciones realmente. Por un lado, se va a chequear la variable que contiene la acción para separar el código de cada una de las acciones. Lo que se debe hacer para cada acción es lo siguiente:

- Consulta:
 - Chequear que se haya metido un código válido (numérico mayor que cero).
 - Buscar en la base de datos la película con el código introducido.
 - Leer los resultados y chequear si hay alguno.
 - Si se ha encontrado, poner el título y la duración en las cajas de texto correspondientes.
 - En caso contrario y para todos los casos de error, sacar los mensajes correspondientes.
- Alta:
 - Se ejecuta la consulta de inserción con los datos de las cajas

4. Programación gráfica

- Si hay algún problema, se muestra el mensaje correspondiente.
- Se vacían las cajas de texto.
- Baja:
 - Se ejecuta la consulta de eliminación usando el código.
 - Si hay algún problema, se muestra el mensaje correspondiente.
 - Se vacían las cajas de texto.
- Modificación:
 - Se ejecuta la consulta de modificación usando el código como condición y los otros datos para cambiar los contenidos.
 - Si hay algún problema, se muestra el mensaje correspondiente.
 - Se vacían las cajas de texto.

El código que realiza todo esto es el siguiente:

```
private void butAceptar_Click(object sender, EventArgs e)
{
    // Se chequea la acción que hay que ejecutar
    if (sAcción == "Consulta")
    {
        // Se chequea si hay algún código metido
        try
        {
            if (Convert.ToInt32(txtCodigo.Text) > 0)
            {
                // Se realiza la consulta
                String sConsulta = "SELECT * "
                    + "FROM Peliculas "
                    + "WHERE Codigo=" + txtCodigo.Text;
                OleDbCommand oleComando = new OleDbCommand(s-
                    Consulta, conBD);
                OleDbDataReader oleReader = oleComando.Exe-
                    cuteReader();
                // Se lee el resultado para ver si hay alguno
                if (oleReader.Read())
                {
                    // Se modifican las cajas de texto con los
                    // datos del registro leído
                    txtTitulo.Text = Convert.To-
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
        txtMinutos.Text = Convert.To-
                        String(oleReader["Duración"]);
                }
                else
                {
                    // Si no hay ningún registro, se indica
                    // con un mensaje
                    MessageBox.Show("El código introducido no
                        existe");
                }
            }
            else
            {
                // Si se ha metido un código negativo, se
                // solicita otro
                MessageBox.Show("Introducir un código válido
                    antes de consultar");
            }
        }
        catch (Exception ex)
        {
            // Si se ha introducido un código no numérico, se
            // solicita otro
            MessageBox.Show("Introducir un código antes de
                consultar");
        }
    }
    else
    {
        if (sAccion == "Alta")
        {
            try
            {
                // Se realiza la inserción
                String sConsulta =
                    "INSERT INTO Peliculas (Título, Dura-
                    ción) "
                    + "VALUES ('" + txtTitulo.Text + "', "
                    + txtMinutos.Text + ")";
                OleDbCommand oleComando = new OleDbCommand();
                oleComando.CommandType = CommandType.Text;
                oleComando.CommandText = sConsulta;
                oleComando.Connection = conBD;
                // Se ejecuta el comando de inserción
                oleComando.ExecuteNonQuery();
                // Se borran las cajas de texto
                txtCodigo.Text = "";
                txtTitulo.Text = "";
                txtMinutos.Text = "";
            }
        }
    }
}
```

4. Programación gráfica

```
        catch (Exception ex)
    {
        // Si se ha producido un error, se muestra un
        // mensaje
        MessageBox.Show("Error en la inserción");
    }
}
else
{
    if (sAccion == "Baja")
    {
        try
        {
            // Se realiza la eliminación
            String sConsulta = "DELETE FROM Peliculas "
                + "WHERE Código=" + txtCodigo.Text;
            OleDbCommand oleComando = new OleDbCom-
                mand();
            oleComando.CommandType = CommandType.Text;
            oleComando.CommandText = sConsulta;
            oleComando.Connection = conBD;
            // Se ejecuta el comando de eliminación
            oleComando.ExecuteNonQuery();
            // Se borran las cajas de texto
            txtCodigo.Text = "";
            txtTitulo.Text = "";
            txtMinutos.Text = "";
        }
        catch (Exception ex)
        {
            // Si se ha producido un error,
            // se muestra un mensaje
            MessageBox.Show("Error en el borrado");
        }
    }
    else
    {
        try
        {
            // Se realiza la modificación
            String sConsulta = "UPDATE Peliculas "
                + "SET Título=' " + txtTitulo.Text + " ', "
                + "Duración=" + txtMinutos.Text + " "
                + "WHERE Código=" + txtCodigo.Text;
            OleDbCommand oleComando = new OleDbCom-
                mand();
            oleComando.CommandType = CommandType.Text;
            oleComando.CommandText = sConsulta;
            oleComando.Connection = conBD;
```

```
        oleComando.ExecuteNonQuery();
        // Se borran las cajas de texto
        txtCodigo.Text = "";
        txtTitulo.Text = "";
        txtMinutos.Text = "";
    }
    catch (Exception ex)
    {
        // Si se ha producido un error, se muestra
        // un mensaje
        MessageBox.Show("Error al modificar");
    }
}
}
}
// Se termina la acción realizada
sAccion = "";
// Se activan y desactivan los botones correspondientes
activarAcciones();
}
```

Práctica paso a paso

Para realizar este completo ejemplo paso a paso se van a poner en práctica los conocimientos adquiridos en todo este capítulo.

En esta práctica se van a tener dos ventanas diferentes que van a permitir gestionar viajes vacacionales y sus diferentes etapas.

Antes de nada se debe diseñar la base de datos que permita gestionar dichos datos. Esta base de datos va a tener dos tablas con los siguientes campos:

- *Viaje* → datos genéricos del viaje:
 - *viald*: identificador.
 - *viaOrigen*: lugar de origen.
 - *viaDestino*: lugar de destino.
 - *viaVehiculo*: vehículo con el que se viaja.

4. Programación gráfica

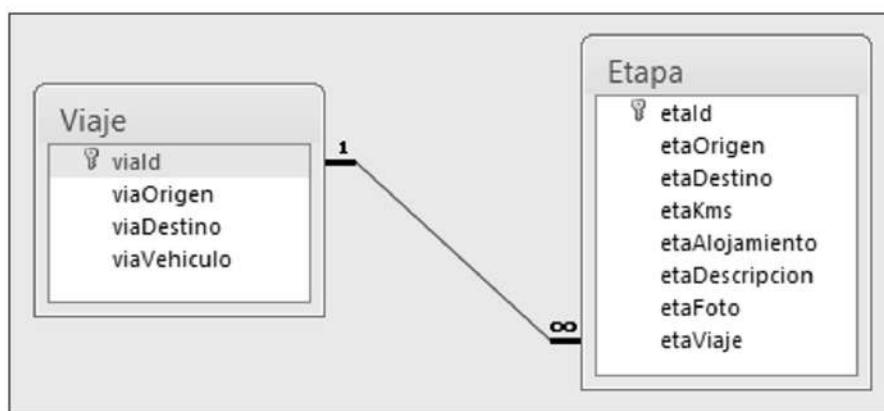
- *Etapa* → datos genéricos de cada etapa del viaje:
 - *etaid*: identificador.
 - *etaOrigen*: lugar de origen.
 - *etaDestino*: lugar de destino.
 - *etaKms*: kilómetros que se realizan.
 - *etaAlojamiento*: opción de alojamiento.
 - *etaDescripcion*: descripción de la etapa.
 - *etaFoto*: fotografía del lugar que se visita.
 - *etaViaje*: viaje al que corresponde esta etapa.

Estas tablas se van a diseñar utilizando Access y el aspecto que van a tener en vista de diseño es como el de las siguientes imágenes:

Viaje		
	Field Name	Data Type
?	viald	AutoNumber
	viaOrigen	Text
	viaDestino	Text
	viaVehiculo	Text

Etapa		
	Field Name	Data Type
?	etaid	AutoNumber
	etaOrigen	Text
	etaDestino	Text
	etaKms	Number
	etaAlojamiento	Text
	etaDescripcion	Text
	etaFoto	Text
	etaViaje	Number

Hay que tener en cuenta que estas dos tablas están relacionadas entre sí, puesto que cada etapa hace referencia a un viaje concreto. Para cada registro de la tabla *Viaje* habrá uno o más registros de la tabla *Etapa*. El campo *viald* debe estar relacionado con el campo *etaViaje*. Para ello hay que ir al apartado de relaciones de *Access*, incluir ambas tablas y establecer la relación entre ellas. El resultado debe ser como el que se puede ver en esta imagen.



El siguiente paso es comenzar con el diseño de la aplicación. Cuando se arranque, se abrirá directamente la ventana de los viajes, así que debe ser la principal y la primera que se va a mostrar. Por ello, lo más lógico es empezar con el diseño de dicha ventana.

En esta ventana se va a tener una lista de viajes. Cada elemento de dicha lista será un viaje de los que hay en la base de datos. El texto que aparecerá en la lista se va a generar a partir de los datos, pero construyendo una frase del tipo:

Viaje de ooooo a dddd (vvvv)

En esta frase, *ooooo* va a ser el origen del viaje, *ddddd* va a ser el destino y *vvvvv* va a ser el vehículo utilizado. La lista no tendrá límite de elementos y aparecerá automáticamente una barra de desplazamiento si hay más viajes de los que se pueden visualizar a la vez.

Aparte de la lista, debajo aparecerán estos mismos datos, pero de manera desglosada, cada uno en una caja de texto diferente y con su propia etiqueta asociada.

4. Programación gráfica

Por último, se van a tener los botones principales para realizar una gestión de datos, como son:

- Alta
- Baja
- Modificación
- Consulta

Estos botones deberán generar la acción correspondiente, pero ésta no se realizará hasta que se pulse un botón para aceptarla, teniendo también otro para cancelarla.

Como se vio en un ejemplo anterior, los botones principales y los de *Aceptar* y *Cancelar* no estarán activos de forma simultánea, así que la pulsación de cada uno de ellos modificará la habilitación o inhabilitación de los botones del grupo contrario.

El funcionamiento para cada una de las acciones debe ser el siguiente:

- Para realizar un alta se llenarán las cajas correspondientes, se pulsará el botón *Alta* y después el botón *Aceptar*. Al hacerlo, el nuevo viaje aparecerá en la lista de viajes con el mismo formato que el resto.
- Para realizar una baja, primero se seleccionará un elemento de la lista, el cual se mostrará directamente en las cajas de texto, después se pulsará el botón *Baja* y, por último, el botón *Aceptar*. En ese momento, el viaje eliminado desaparecerá de la lista de viajes.
- Para hacer una modificación en un viaje, primero se seleccionará el elemento de la lista. Una vez que aparezca el contenido en las cajas de texto, se podrán modificar los campos que se deseen. Una vez realizados los cambios, se pulsará el botón *Modificación* y se terminará pulsando el botón *Aceptar*. Al hacerlo, deberá verse en la lista el registro seleccionado con los nuevos datos.
- Para hacer una consulta se empezará por seleccionar de la lista el viaje que se quiera consultar, cuyos datos se mostrarán en las cajas de texto inferiores. Después se pulsará el botón *Consulta* y se acabará pulsando el botón *Aceptar*. En ese momento se abrirá la segunda ventana para ver las etapas relacionadas con el viaje consultado.

Por último, y ya que es la ventana principal, habrá también un botón para finalizar el programa.

El diseño de esta ventana puede tener un aspecto como el que se muestra en la imagen siguiente:



Para realizar esta ventana se va a empezar declarando las variables generales. Como en otras ocasiones, se necesita una variable para la conexión

4. Programación gráfica

con la base de datos y otra variable para la acción que se va a gestionar. Se deben declarar así:

```
// Variables principales
System.Data.OleDb.OleDbConnection conBD = new System.Data.
    OleDb.OleDbConnection();
String sAccion = "";
```

Se deben crear un par de funciones para abrir la conexión con la base de datos y para cerrar dicha conexión. Como ya se han explicado en un ejemplo anterior, simplemente se va a poner aquí el código de estas funciones:

```
public void abrirConexion()
{
    // Se crea el string de conexión a la base de datos
    conBD.ConnectionString = @"Provider=Microsoft.ACE.
        OLEDB.12.0; Data Source=..\BD\Viajes.accdb; Persist
        Security Info=False";
    try
    {
        // Se abre la base de datos
        conBD.Open();
    }
    catch (Exception ex)
    {
        MessageBox.Show("Error de conexión" + ex);
    }
}

public void cerrarConexion()
{
    // Se chequea si está la conexión abierta
    if (conBD.State == ConnectionState.Open)
    {
        // Se cierra la conexión
        conBD.Close();
    }
}
```

Asimismo, se ha explicado en otro ejemplo de este mismo capítulo cómo habilitar e inhabilitar los grupos de botones correspondientes, así que

simplemente se va a poner aquí el código de las dos funciones que van a realizar estas dos acciones:

```
private void desactivarAcciones()
{
    butConsulta.Enabled = false;
    butAlta.Enabled = false;
    butBaja.Enabled = false;
    butModificacion.Enabled = false;
    butAceptar.Enabled = true;
    butCancelar.Enabled = true;
}

private void activarAcciones()
{
    butConsulta.Enabled = true;
    butAlta.Enabled = true;
    butBaja.Enabled = true;
    butModificacion.Enabled = true;
    butAceptar.Enabled = false;
    butCancelar.Enabled = false;
    // Se cancela la acción que hay que realizar
    sAccion = "";
}
```

Cuando arranca la aplicación, lo primero que se debe hacer es llenar la lista de viajes con los que se tengan en la base de datos. Para ello se puede crear una función específica que lea los datos de la tabla de viajes y vaya generando los elementos para la tabla.

En esta función se va a empezar vaciando la lista de elementos, después se realizará la consulta a la base de datos y en un bucle se generará cada una de las líneas usando una variable de texto en la que se vayan concatenando las partes de texto fijas con los campos de la tabla de viajes. Esta función puede ser la siguiente:

4. Programación gráfica

```
private void leerViajes()
{
    try
    {
        // Se vacía la lista
        lstViajes.Items.Clear();
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM Viaje";
        OleDbCommand oleComando = new OleDbCommand(sCon-
sulta,
            conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno
        while (oleReader.Read())
        {
            // Se genera la línea con los datos del viaje
            String sLinea = "Viaje de ";
            sLinea = sLinea +
                Convert.ToString(oleReader["viaOrigen"]);
            sLinea = sLinea + " a ";
            sLinea = sLinea +
                Convert.ToString(oleReader["viaDestino"]);
            sLinea = sLinea + "(";
            sLinea = sLinea +
                Convert.ToString(oleReader["viaVehiculo"]);
            sLinea = sLinea + ")";
            // Se añade la línea a la lista
            lstViajes.Items.Add(sLinea);
        }
    }
    catch (Exception ex)
    {
        // Si salta una excepción, se muestra el error
        MessageBox.Show("Error: " + ex);
    }
}
```

Cuando arranca la aplicación se ejecuta el evento *Load* del formulario y en ese evento se va a llamar a las funciones de abrir conexión y a la de llenar la lista con los viajes. Debe quedar así:

```
private void frmEjercicio18_Load(object sender, EventArgs
e)
{
    // Se abre la conexión con la base de datos
    abrirConexion();
    leerViajes();
}
```

Una vez que esté cargada la lista, una de las opciones que se tienen es pulsar en uno de los elementos. Al hacerlo, se muestran en las cajas de texto inferiores los datos que corresponden al viaje seleccionado. El evento que se va a producir cuando al pulsar sobre un elemento de la lista es *SelectedIndexChanged*, puesto que lo que ocurre es que cambia el índice de selección.

Siendo así, lo que se va a programar en dicho evento es la separación del texto de la línea completa en las partes correspondientes. Para ello se van a usar dos métodos:

- *Substring*: devuelve una parte de una cadena indicando la posición del carácter inicial de la subcadena y su longitud.
- *IndexOf*: dada una subcadena, devuelve la posición en la que se encuentra dentro de la cadena principal.

Para obtener el elemento que está seleccionado en la lista se debe usar el método *SelectedItem* y después convertirlo a *String*, puesto que lo que se devuelve siempre es un objeto.

La función que realiza estas acciones es la siguiente:

```
private void lstViajes_SelectedIndexChanged(object sender,
                                         EventArgs e)
{
    // Se obtienen los datos del elemento seleccionado
    String sViaje = lstViajes.SelectedItem.ToString();
    sViaje = sViaje.Substring(9);
    txtOrigen.Text = sViaje.Substring(0, sViaje.IndexOf(
        " a "));
    sViaje = sViaje.Substring(sViaje.IndexOf(" a ") + 3);
    txtDestino.Text = sViaje.Substring(0, sViaje.IndexOf(
        "("));
    sViaje = sViaje.Substring(sViaje.IndexOf("(") + 2);
    txtVehiculo.Text = sViaje.Substring(0,
        sViaje.IndexOf(")")));
}
```

Los botones principales únicamente van a establecer la acción correspondiente y van a habilitar los botones de *Aceptar* y *Cancelar*, aparte de des-

4. Programación gráfica

habilitarse a ellos mismos. Su código es similar para los cuatro casos y muy sencillo, tal y como se puede apreciar en estas funciones:

```
private void butAlta_Click(object sender, EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAccion = "Alta";
    // Se activan y desactivan los botones correspondientes
    desactivarAcciones();
}

private void butBaja_Click(object sender, EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAccion = "Baja";
    // Se activan y desactivan los botones correspondientes
    desactivarAcciones();
}

private void butModificacion_Click(object sender, EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAccion = "Modificación";
    // Se activan y desactivan los botones correspondientes
    desactivarAcciones();
}

private void butConsulta_Click(object sender, EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAccion = "Consulta";
    // Se activan y desactivan los botones correspondientes
    desactivarAcciones();
}
```

Antes de ver las funciones de los botones que quedan, conviene realizar otra función auxiliar que viene muy bien, puesto que se va a utilizar en varios de los casos. Se trata de una función que, dados los datos que se encuentran en la lista de viajes, sea capaz de devolver el código del viaje al que corresponden. Esta función debe realizar la extracción de los campos de la misma manera que cuando se selecciona el elemento para colocar en las cajas de texto. Una vez que se tengan los datos en sendas varia-

bles, se realizará la consulta a la base de datos con las tres condiciones y se obtendrá el código que debe ser devuelto al finalizar la función. Esta función puede ser como se muestra aquí:

```
private int leerCodigo()
{
    int iCodigo = -1;
    try
    {
        // Se obtienen los datos de la lista
        String sViaje = lstViajes.SelectedItem.ToString();
        sViaje = sViaje.Substring(9);
        String sOrigen = sViaje.Substring(0,
            sViaje.IndexOf(" a "));
        sViaje = sViaje.Substring(sViaje.IndexOf(" a ") + 3);
        String sDestino = sViaje.Substring(0,
            sViaje.IndexOf("("));
        sViaje = sViaje.Substring(sViaje.IndexOf("(") + 2);
        String sVehiculo = sViaje.Substring(0,
            sViaje.IndexOf(")"));
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM Viaje "
            + "WHERE viaOrigen= '" + sOrigen + "' "
            + "AND viaDestino= '" + sDestino + "' "
            + "AND viaVehiculo= '" + sVehiculo + "' ";
        OleDbCommand oleComando = new OleDbCommand(sConsulta,
            conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno
        if (oleReader.Read())
        {
            // Se obtiene el código
            iCodigo = Convert.ToInt32(oleReader["viaId"]);
        }
    }
    catch (Exception ex)
    {
    }
    return iCodigo;
}
```

El botón con el código más complejo es el botón *Aceptar*, que debe realizar las siguientes instrucciones en función de la acción seleccionada:

4. Programación gráfica

- *Consulta:* Se comienza obteniendo el código del viaje, se crea la ventana para desplegar las etapas, se le pasan los datos que van a ser requeridos en la nueva ventana de etapas (código del viaje y conexión a la base de datos) y se termina abriendo la nueva ventana.
- *Alta:* Se crea la consulta de inserción usando los datos de las cajas de texto, se ejecuta dicha consulta, se vacían las cajas de texto y se actualiza la lista de viajes para que aparezca el nuevo.
- *Baja:* Se comienza obteniendo el código del viaje, se crea la consulta de eliminación usando ese código, se ejecuta dicha consulta, se vacían las cajas de texto y se actualiza la lista de viajes para que desaparezca el eliminado.
- *Modificación:* Se comienza obteniendo el código del viaje, se crea la consulta de modificación usando ese código y las cajas de texto, se ejecuta dicha consulta, se vacían las cajas de texto y se actualiza la lista de viajes para que aparezca la información modificada.

El evento debe terminar vaciando la variable de la acción y habilitando y deshabilitando los botones correspondientes.

El método *Click* del botón queda así:

```
private void butAceptar_Click(object sender, EventArgs e)
{
    // Se chequea la acción que hay que ejecutar
    if (sAcción == "Consulta")
    {
        // Se chequea si hay algún código metido
        try
        {
            // Se obtiene el código del viaje
            int iCodigo = leerCodigo();
            // Se abre la ventana de etapas
            frmEtapas frmVentana = new frmEtapas();
            // Se asigna el código de viaje a visualizar
            frmVentana.iViaje = iCodigo;
            // Se pasa también la conexión a la BD para
            // reutilizarla
            frmVentana.conBD = conBD;
            // Se muestra la ventana
            frmVentana.ShowDialog();
        }
    }
}
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
{  
    // Si se ha introducido un código no numérico, se  
    // solicita otro  
    MessageBox.Show("Introducir un código antes de  
    consultar");  
}  
}  
else  
{  
    if (sAccion == "Alta")  
    {  
        try  
        {  
            // Se realiza la inserción  
            String sConsulta =  
                "INSERT INTO Viaje (viaOrigen, "  
                + "viaDestino, " viaVehiculo) "  
                + "VALUES ('" + txtOrigen.Text + "', '"  
                + txtDestino.Text + "','"  
                + txtVehiculo.Text + "')";  
            OleDbCommand oleComando = new OleDbCommand();  
            oleComando.CommandType = CommandType.Text;  
            oleComando.CommandText = sConsulta;  
            oleComando.Connection = conBD;  
            // Se ejecuta el comando de inserción  
            oleComando.ExecuteNonQuery();  
            // Se borran las cajas de texto  
            txtOrigen.Text = "";  
            txtDestino.Text = "";  
            txtVehiculo.Text = "";  
            // Se actualiza la lista de viajes  
            leerViajes();  
        }  
        catch (Exception ex)  
        {  
            // Si se ha producido un error, se muestra un  
            // mensaje  
            MessageBox.Show("Error en la inserción: " + ex);  
        }  
    }  
    else  
    {  
        if (sAccion == "Baja")  
        {  
            try  
            {  
                // Se obtiene el código del viaje  
                int iCodigo = leerCodigo();  
            }  
        }  
    }  
}
```

4. Programación gráfica

```
String sConsulta = "DELETE FROM Viaje "
    + "WHERE viaId=" + iCodigo;
OleDbCommand oleComando = new OleDbCommand();
oleComando.CommandType = CommandType.Text;
oleComando.CommandText = sConsulta;
oleComando.Connection = conBD;
// Se ejecuta el comando de eliminación
oleComando.ExecuteNonQuery();
// Se borran las cajas de texto
txtOrigen.Text = "";
txtDestino.Text = "";
txtVehiculo.Text = "";
// Se actualiza la lista de viajes
leerViajes();
}
catch (Exception ex)
{
    // Si se ha producido un error, se muestra
    // un mensaje
    MessageBox.Show("Error en la eliminación: "
        + ex);
}
else
{
    try
    {
        // Se obtiene el código del viaje
        int iCodigo = leerCodigo();
        // Se realiza la modificación
        String sConsulta = "UPDATE Viaje "
            + "SET viaOrigen=' " + txtOrigen.Text
            + "' , " + "viaDestino=' "
            + txtDestino.Text + "' , "
            + "viaVehiculo=' " + txtVehiculo.Text
            + "' "
            + "WHERE viaId=" + iCodigo;
        OleDbCommand oleComando = new OleDbCommand();
        oleComando.CommandType = CommandType.Text;
        oleComando.CommandText = sConsulta;
        oleComando.Connection = conBD;
        // Se ejecuta el comando de modificación
        oleComando.ExecuteNonQuery();
        // Se borran las cajas de texto
        txtOrigen.Text = "";
        txtDestino.Text = "";
        txtVehiculo.Text = "";
    }
}
```

```
        leerViajes();
    }
    catch (Exception ex)
    {
        // Si se ha producido un error, se muestra
        // un mensaje
        MessageBox.Show("Error al modificar");
    }
}
}
// Se termina la acción realizada
sAccion = "";
// Se activan y desactivan los botones correspondientes
activarAcciones();
}
```

El siguiente botón es el de *Cancelar*, cuyo código es muy sencillo, porque lo único que debe hacer es habilitar y deshabilitar los botones correspondientes. También se puede vaciar la variable de acción seleccionada, aunque no es imprescindible, puesto que cuando se pulse un nuevo botón de acción, se va a reemplazar lo que pudiera tener. El código de este botón queda así:

```
private void butCancelar_Click(object sender, EventArgs e)
{
    // Se termina la acción realizada
    sAccion = "";
    // Se activan y desactivan los botones correspondientes
    activarAcciones();
}
```

El último botón de esta ventana es el de *Salir*, que como siempre debe cerrar la conexión con la base de datos y cerrar la ventana principal. El código es:

```
private void butSalir_Click(object sender, EventArgs e)
{
    // Se cierra la conexión antes de terminar
    cerrarConexion();
    // Finaliza el programa
    this.Close();
}
```

4. Programación gráfica

La segunda ventana es similar a la primera, aunque se le ha añadido alguna complejidad más. A la hora de diseñar esta nueva ventana hay que tener en cuenta todo lo que debe cumplir.

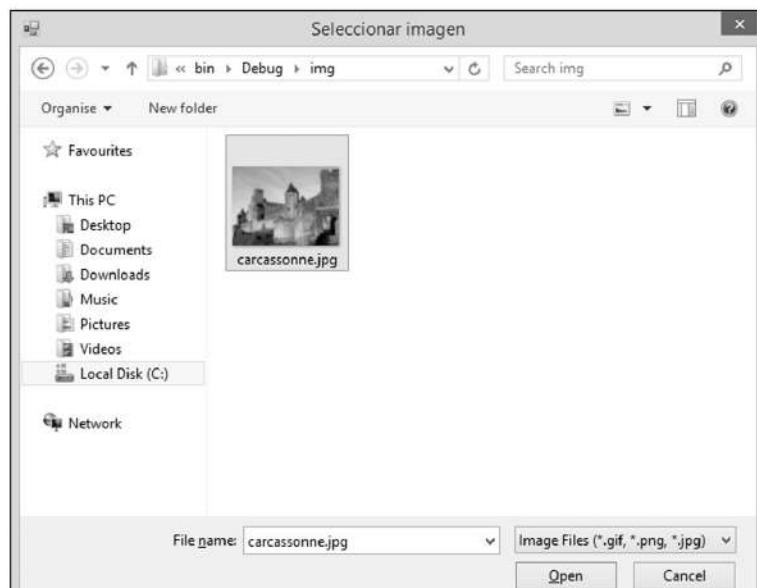
En esta ventana se va a tener una lista de etapas. Cada elemento de dicha lista será una etapa de las que hay en la base de datos y pertenecerá al viaje que ha sido seleccionado en la ventana anterior. El texto que aparecerá en la lista se va a generar a partir de los datos, pero con una frase del tipo:

Etapa de ooooo a ddddd (kkkkk Km)

En esta frase, *ooooo* va a ser el origen de la etapa, *ddddd* va a ser el destino y *kkkkk* van a ser los kilómetros que tenga la etapa. La lista no tendrá límite de elementos y aparecerá automáticamente una barra de desplazamiento si hay más etapas de las que se pueden visualizar a la vez.

Aparte de la lista, debajo aparecerán estos mismos datos y el resto que hay en la tabla de etapas, pero de manera desglosada, cada uno en una caja de texto diferente y con su propia etiqueta asociada, además de mostrar la imagen asociada a la etapa en una *PictureBox*.

Se debe poder pulsar en cualquier momento sobre la imagen y, al hacerlo, saldrá un cuadro de diálogo para buscar la imagen. El aspecto de este cuadro de diálogo será como el de esta imagen:



Una vez que se haya seleccionado la imagen, se va a guardar en una variable global para su posterior uso. De esta forma, cuando se tenga que hacer una modificación o se vaya a añadir una etapa, se podrá usar directamente el valor contenido en esta variable.

Por último, se van a tener los botones principales para realizar una gestión de datos como son:

- Alta.
- Baja.
- Modificación.

Estos botones deberán generar la acción correspondiente, pero ésta no se realizará hasta que se pulse un botón para aceptarla, teniendo también otro para cancelarla.

De la misma forma que en la ventana de viajes, los botones principales y los de *Aceptar* y *Cancelar* no estarán activos de forma simultánea, así que la pulsación de cada uno de ellos modificará la habilitación o inhabilitación de los botones del grupo contrario.

El funcionamiento para cada una de las acciones debe ser el siguiente:

- Para realizar un alta se llenarán las cajas correspondientes, se pulsará el botón *Alta* y después el botón *Aceptar*. Al hacerlo, la nueva etapa aparecerá en la lista de etapas con el mismo formato que el resto.
- Para realizar una baja, primero se seleccionará un elemento de la lista, el cual se mostrará directamente en las cajas de texto y en la imagen, después se pulsará el botón *Baja* y, por último, el botón *Aceptar*. En ese momento la etapa eliminada desaparecerá de la lista de etapas.
- Para hacer una modificación en una etapa, primero se seleccionará el elemento de la lista. Cuando aparezca el contenido en las cajas de texto, se podrán modificar los campos que se deseen, incluida la imagen. Una vez realizados los cambios, se pulsará el botón *Modificación* y se terminará pulsando el botón *Aceptar*. Al hacerlo, deberá verse en la lista el registro seleccionado con los nuevos datos.

Por último, y ya que es la ventana secundaria, habrá también un botón

4. Programación gráfica

El diseño de esta segunda ventana puede tener un aspecto como el que se muestra en la imagen siguiente:



El código debe comenzar con las variables globales que se van a usar en el programa. A la variable de la conexión con la base de datos y la de la

el identificador del viaje y la que contendrá la ruta de la foto que se está viendo. Cabe destacar que la de la conexión y la del viaje deben ser públicas, puesto que sus valores se van a actualizar desde la otra ventana. Las declaraciones deben ser así:

```
// Variables principales
public System.Data.OleDb.OleDbConnection conBD = new System.
    Data.OleDb.OleDbConnection();
public int iViaje = 0;
String sAccion = "";
String sRutaFoto = "";
```

Para llenar la lista conviene hacer una función específica que obtenga los datos de la tabla correspondiente, utilizando como filtro el identificador del viaje en curso. Dicha función puede ser así:

```
private void leerEtapas()
{
    try
    {
        // Se vacía la lista
        lstEtapas.Items.Clear();
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM Etapa "
            + "WHERE etaViaje=" + iViaje + " "
            + "ORDER BY etaId";
        OleDbCommand oleComando = new OleDbCommand(sConsulta,
            conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno
        while (oleReader.Read())
        {
            // Se genera la línea con los datos del viaje
            String sLinea = "Etapa de ";
            sLinea = sLinea
                + Convert.ToString(oleReader["etaOrigen"]);
            sLinea = sLinea + " a ";
            sLinea = sLinea
```

4. Programación gráfica

```
        + Convert.ToString(oleReader["etaDestino"]);
    sLinea = sLinea + " (";
    sLinea = sLinea
        + Convert.ToString(oleReader["etaKms"]);
    sLinea = sLinea + " Kms)";
    // Se añade la línea a la lista
    lstEtapas.Items.Add(sLinea);
}
}
catch (Exception ex)
{
    // Si salta una excepción, se muestra el error
    MessageBox.Show("Error: " + ex);
}
}
```

Al arrancar esta ventana se llamará a esta función para que desde el inicio se vean las etapas del viaje que se está consultando. Se hará así:

```
private void frmEtapas_Load(object sender, EventArgs e)
{
    // Se leen las etapas que se mostrarán en la lista
    leerEtapas();
}
```

Las funciones para habilitar y deshabilitar botones son iguales que las de la ventana anterior, quitando el botón de consulta, que ya no existe. No merece la pena repetir el código aquí, puesto que es muy similar.

De la misma forma, las instrucciones para los botones principales son idénticas a las de la ventana principal, puesto que sólo deben establecer la acción y habilitar y deshabilitar los botones correspondientes. Basta con copiar y pegar las mismas funciones desde la otra ventana.

Por supuesto, el botón para cancelar la acción también va a ser idéntico, así que no hace falta reproducirlo de nuevo.

De la misma forma que en la ventana anterior para los viajes, es recomendable tener una función que obtenga el código de una etapa a partir de los datos que se tienen en la lista de elementos. Esta función va a ser bastante similar a la anterior, pero modificando los textos fijos de esta ventana a la hora de extraer los elementos, ya que son algo diferentes.

Después se debe hacer la consulta a la tabla de etapas y obtener su código. Las instrucciones de esta función serán:

```
private int leerCodigo()
{
    int iCodigo = -1;
    // Se chequea si hay algún código metido
    try
    {
        // Se obtienen los datos de la lista
        String sEtapa = lstEtapas.SelectedItem.ToString();
        sEtapa = sEtapa.Substring(9);
        String sOrigen = sEtapa.Substring(0,
            sEtapa.IndexOf(" a "));
        sEtapa = sEtapa.Substring(sEtapa.IndexOf(" a ") + 3);
        String sDestino = sEtapa.Substring(0,
            sEtapa.IndexOf(" ("));
        sEtapa = sEtapa.Substring(sEtapa.IndexOf(" (") + 2);
        String sKms = sEtapa.Substring(0,
            sEtapa.IndexOf(" Kms")));
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM Etapa "
            + "WHERE etaOrigen= '" + sOrigen + "' "
            + "AND etaDestino= '" + sDestino + "' "
            + "AND etaKms= '" + sKms;
        OleDbCommand oleComando = new OleDbCommand(sConsulta,
            conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno
        if (oleReader.Read())
        {
            // Se obtiene el código
            iCodigo = Convert.ToInt32(oleReader["etaId"]);
        }
    }
    catch (Exception ex)
    {
    }
    return iCodigo;
}
```

Cuando se pulse en uno de los elementos de la lista, también hay que hacer algo similar a lo que se hacía con los viajes. En este caso, además de llenar las cajas de texto correspondientes, se va a modificar el con-

4. Programación gráfica

tenido de la imagen. Como las cajas de texto visualizan más datos que los que hay en la lista, lo que se debe hacer es sacarlos directamente de la tabla de etapas de la base de datos.

Para modificar la imagen, primero se va a guardar la ruta completa en la variable global destinada a tal efecto. Una vez que se tenga dicha variable rellena con la ruta, se modificará la propiedad *Image* con la imagen que se obtenga utilizando el método *Image.FromFile* al que se le pasa la ruta. Si no hubiera ninguna ruta asociada, entonces la imagen se deberá dejar vacía. Para ello habrá que asignar en la misma propiedad el valor nulo.

Las acciones completas para este evento van a ser:

```
private void lstEtapas_SelectedIndexChanged(object sender,
EventArgs e)
{
    // Se busca primero el código de la etapa
    int iCodigo = leerCodigo();
    // Se obtienen los datos del elemento seleccionado
    try
    {
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM Etapa "
            + "WHERE etaId= " + iCodigo;
        OleDbCommand oleComando = new OleDbCommand(sConsulta,
            conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno
        if (oleReader.Read())
        {
            // Se colocan los datos
            txtOrigen.Text =
                Convert.ToString(oleReader["etaOrigen"]);
            txtDestino.Text =
                Convert.ToString(oleReader["etaDestino"]);
            txtKilometros.Text =
                Convert.ToString(oleReader["etaKms"]);
            txtAlojamiento.Text =
                Convert.ToString(oleReader["etaAlojamiento"]);
            txtDescripcion.Text =
                Convert.ToString(oleReader["etaDescripcion"]);
            sRutaFoto =
                Convert.ToString(oleReader["etaFoto"]);
            if (sRutaFoto.Length>0)
            {

```

```
// Se muestra la imagen
picFoto.Image = Image.FromFile(sRutaFoto);
}
else
{
    // Se vacía la imagen
    picFoto.Image = null;
}
}
}
catch (Exception ex)
{
}
}
```

Cuando se pulse sobre la imagen, tendrá que aparecer un cuadro de diálogo para elegir la ruta y el archivo de la imagen deseada. Antes de sacar el cuadro de texto se pueden modificar varios aspectos del mismo, como:

- *Title*: título del cuadro de diálogo.
- *InitialDirectory*: carpeta desde la que empezar a buscar.
- *Filter*: filtros para las extensiones.

Una vez conseguido el aspecto deseado, se muestra el cuadro de diálogo. Si se ha elegido un archivo, se guarda la ruta en la variable global y también se modifica la imagen con el nuevo archivo escogido. Se debe hacer de esta manera:

```
private void picFoto_Click(object sender, EventArgs e)
{
    // Se pide la imagen con el cuadro de diálogo
    OpenFileDialog opeDialogo = new OpenFileDialog();
    opeDialogo.Title = "Seleccionar imagen";
    opeDialogo.InitialDirectory = @"C:\";
    opeDialogo.Filter =
        "Image Files (*.gif, *.png, *.jpg) | *.gif; *.png; *.
jpg";
    if (opeDialogo.ShowDialog() == DialogResult.OK)
    {
        // Se guarda la imagen seleccionada
        sRutaFoto = opeDialogo.FileName;
        // Se muestra la imagen
        picFoto.Image = Image.FromFile(sRutaFoto);
    }
}
```

4. Programación gráfica

Y de nuevo el evento con mayor complejidad es el botón para aceptar la acción escogida. Las acciones de este botón son tres, puesto que en esta ventana no existe la opción de consultar, ya que las consultas de cada etapa se hacen directamente pulsando el elemento deseado sobre la lista.

Para las acciones de *Añadir*, *Eliminar* y *Modificar*, las instrucciones son básicamente las mismas que en la ventana principal, modificando la tabla y los campos correspondientes. Lo que sí se debe tener en cuenta es que el identificador del viaje debe usarse como filtro y también que la ruta de la foto se obtiene de la variable global, que siempre estará actualizada.

El código completo para este evento debe ser como el mostrado a continuación:

```
private void butAceptar_Click(object sender, EventArgs e)
{
    // Se chequea la acción que hay que ejecutar
    if (sAcción == "Alta")
    {
        try
        {
            // Se realiza la inserción
            String sConsulta =
                "INSERT INTO Etapa (etaOrigen, etaDestino, "
                + "etaKms, etaAlojamiento, etaDescripción, "
                + "etaFoto, etaViaje) "
                + "VALUES ('" + txtOrigen.Text + "', ''"
                + txtDestino.Text + "', ''"
                + txtKilometros.Text + "', ''"
                + txtAlojamiento.Text + "', ''"
                + txtDescripción.Text + "', '" + sRutaFoto
                + "', '" + iViaje + "')";
            OleDbCommand oleComando = new OleDbCommand();
            oleComando.CommandType = CommandType.Text;
            oleComando.CommandText = sConsulta;
            oleComando.Connection = conBD;
            // Se ejecuta el comando de inserción
            oleComando.ExecuteNonQuery();
            // Se borran las cajas de texto
            txtOrigen.Text = "";
            txtDestino.Text = "";
            txtKilometros.Text = "";
            txtAlojamiento.Text = "";
            txtDescripción.Text = "";
            sRutaFoto = "";
            picFoto.Image = null;
            // Se actualiza la lista de etapas
        }
    }
}
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
        leerEtapas();
    }
    catch (Exception ex)
    {
        // Si se ha producido un error, se muestra un
        // mensaje
        MessageBox.Show("Error en la inserción: " + ex);
    }
}
else
{
    if (sAccion == "Baja")
    {
        try
        {
            // Se obtiene el código de la etapa
            int iCodigo = leerCodigo();
            // Se realiza la eliminación
            String sConsulta = "DELETE FROM Etapa "
                + "WHERE etaId=" + iCodigo;
            OleDbCommand oleComando = new OleDbCommand();
            oleComando.CommandType = CommandType.Text;
            oleComando.CommandText = sConsulta;
            oleComando.Connection = conBD;
            // Se ejecuta el comando de eliminación
            oleComando.ExecuteNonQuery();
            // Se borran las cajas de texto
            txtOrigen.Text = "";
            txtDestino.Text = "";
            txtKilometros.Text = "";
            txtAlojamiento.Text = "";
            txtDescripcion.Text = "";
            sRutaFoto = "";
            picFoto.Image = null;
            // Se actualiza la lista de etapas
            leerEtapas();
        }
        catch (Exception ex)
        {
            // Si se ha producido un error, se muestra un
            // mensaje
            MessageBox.Show("Error en la eliminación: " +
                ex);
        }
    }
}
```

4. Programación gráfica

```
else
{
    try
    {
        // Se obtiene el código de la etapa
        int iCodigo = leerCodigo();
        // Se realiza la modificación
        String sConsulta = "UPDATE Etapa "
            + "SET etaOrigen=' " + txtOrigen.Text
            + " ', " + "etaDestino=' "
            + txtDestino.Text + " ', " + "etaKms="
            + txtKilometros.Text + " , "
            + "etaAlojamiento=' " + txtAlojamiento.
Text
            + " ', " + "etaDescripcion=' "
            + txtDescripcion.Text + " ', "
            + "etaFoto=' " + sRutaFoto + " ' "
            + "WHERE etaId=" + iCodigo;
        OleDbCommand oleComando = new OleDbCommand();
        oleComando.CommandType = CommandType.Text;
        oleComando.CommandText = sConsulta;
        oleComando.Connection = conBD;
        // Se ejecuta el comando de modificación
        oleComando.ExecuteNonQuery();
        // Se borran las cajas de texto
        txtOrigen.Text = "";
        txtDestino.Text = "";
        txtKilometros.Text = "";
        txtAlojamiento.Text = "";
        txtDescripcion.Text = "";
        sRutaFoto = "";
        picFoto.Image = null;
        // Se actualiza la lista de etapas
        leerEtapas();
    }
    catch (Exception ex)
    {
        // Si se ha producido un error, se muestra
        // un mensaje
        MessageBox.Show("Error al modificar");
    }
}
// Se termina la acción realizada
sAccion = "";
// Se activan y desactivan los botones correspondientes
activarAcciones();
}
```

El botón *Volver* sólo tiene que cerrar la ventana de esta manera:

```
private void butVolver_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Ejercicios

Ejercicio 4.1

Se trata de hacer una calculadora que tenga las operaciones básicas. El programa debe cumplir las siguientes condiciones:

- Habrá un botón para cada número con la disposición habitual de una calculadora. Según se vayan pulsando estos botones, se irá creando el primer o el segundo operando, según corresponda. También se tendrá el punto decimal para crear los operandos con decimales.
- Habrá un botón para cada una de las cuatro operaciones básicas que permitirán sumar, restar, multiplicar y dividir. Cuando se pulse un botón de operación, los números pasarán a escribirse en el segundo operando.
- Habrá un botón con el signo igual para obtener el resultado de la operación.
- También habrá un botón denominado *Nueva operación* para volver a comenzar otro cálculo.
- Por último, habrá un botón llamado *Salir* que finalizará el programa.

Para evitar errores, habrá que controlar bien qué botones se pueden pulsar en cada momento. Por ejemplo, cuando se pulse un punto en un operando, se deberá desactivar dicho botón para que no se pueda poner otro separador decimal en el mismo operando.

Cuando comience la aplicación deberá aparecer un 0 en cada uno de los dos operandos. Habrá que tenerlo en cuenta para que después al pulsar los dígitos se controle que el número no tenga errores o lo borre.

4. Programación gráfica

El aspecto de la ventana deberá ser como el de la siguiente imagen.



Ejercicio 4.2

Se trata de hacer una aplicación que permita colocar tres imágenes en diferentes lugares de la ventana y que además permita modificar el color del fondo de la ventana principal.

En la parte de arriba de la ventana van a estar los controles y en la parte de abajo van a estar las imágenes, que se pueden recolocar.

Para mover las imágenes a un lugar concreto, lo que se debe hacer es pulsar con el botón izquierdo del ratón en el lugar deseado. Antes se debe indicar qué imagen se quiere modificar, para lo cual en la zona de

control se tendrán unos botones de opción que permitirán la selección de la imagen asociada.

Para modificar el color del fondo se tendrán en la zona de control unas barras para seleccionar la mezcla de colores.

También debe haber un botón para salir de la aplicación en la zona de control.

El aspecto de la ventana debe ser como el de esta imagen:



Ejercicio 4.3

En este ejercicio se trata de hacer una aplicación que permita obtener los actores que han participado en una película. Para ello, hay que crear una base de datos que tenga dos tablas, una para las películas y otra para los actores.

4. Programación gráfica

La aplicación debe tener una caja combinada en la que aparezcan todos los títulos de las películas existentes y una lista con los actores y actrices. Cuando se seleccione un elemento de la lista de películas, éste aparecerá como texto principal de la caja y se actualizarán los datos en la lista de actores para que aparezcan los correspondientes a dicho título.

También debe haber un botón para finalizar la ejecución.

El aspecto final de la ventana deberá ser como el de la siguiente imagen.



Ejercicio 4.4

En este ejercicio se trata de hacer una aplicación que realice pronósticos de los resultados de los partidos de baloncesto. Para calcular los pronósticos se van a tener en cuenta los resultados de los cinco últimos partidos que se hayan jugado entre dos equipos. El objetivo es realizar una media ponderada, dando más valor a los resultados de los partidos más cercanos. Para ello, lo que se va a hacer es realizar una multiplicación de cada valor por 1, 1.1, 1.2, 1.3 y 1.4, empezando por el resultado más lejano en el tiempo y terminando por el más reciente. Para calcular la media ponderada, habrá que dividir el resultado entre los 5 valores y también entre 1.2, que es la media de los valores que se han usado para ponderar.

El programa debe obtener para cada partido el número de puntos totales y el número de puntos ponderado. Además, en el resultado global sacará

la media de puntos que podrá sacar cada equipo y también el total en el partido. Aparecerá también la media ponderada de forma informativa.

Cada vez que se modifique un elemento de una caja de texto, se recalculará todo.

Por último, el color con el que se muestren los resultados de puntuación media será diferente para cada equipo. Si los dos valores son iguales, estarán en color negro. Si los valores son diferentes, el mayor valor se mostrará en color verde y el menor en color rojo.

La ventana que se debe diseñar para la aplicación tiene que tener un aspecto como el de la siguiente imagen.

The screenshot shows a Windows application window titled "Borja Orbegozo". The main title bar has the application name. Below it, the window title is "Predicción de resultados para partidos de baloncesto". The content area contains a table with five rows for game results and one row for averages. The table has four columns: "Equipo 1", "Equipo 2", "Total", and "Total ponderado". Each column has a header cell above the first row. The first four rows are labeled "Resultado partido 1" through "Resultado partido 5". The last row is labeled "Medias:". All cells in the "Equipo 1" and "Equipo 2" columns contain the value "0". The "Total" and "Total ponderado" columns also contain "0". At the bottom center of the window is a button labeled "Salir".

	Equipo 1	Equipo 2	Total	Total ponderado
Resultado partido 1:	0	0	0	0
Resultado partido 2:	0	0	0	0
Resultado partido 3:	0	0	0	0
Resultado partido 4:	0	0	0	0
Resultado partido 5:	0	0	0	0
Medias:	0	0	0	0

Nota: Todos los valores deben ser números enteros.
Hay que eliminar los números decimales resultantes
de las operaciones.

Proyectos completos de programación

CAPÍTULO
5

Proyecto completo con formularios

En este punto se va a realizar un proyecto completo desde cero, aplicando todo lo aprendido a lo largo del libro. El objetivo es realizar un programa de contabilidad casera. Dicho programa permitirá apuntar los ingresos y gastos más habituales en una casa y ver dichos datos acumulados día a día y mes a mes, usando para ello tablas y gráficas. En las gráficas incluso se podrán ver comparativas con los gastos del año anterior y también se mostrará la media ideal para controlar si los gastos del mes van sobre lo previsto o no. Los gastos van a estar organizados por conceptos generales y cada concepto tendrá a su vez una serie de subconceptos.

Para el control se van a utilizar cinco tablas en una base de datos que para este ejemplo se va a llamar *Contabilidad.accdb* y que va a estar hecha en Access. Las tablas necesarias son las siguientes:

- *tabConcepto* → tabla predefinida que va a contener los conceptos por los que se clasifican los gastos.
- *tabSubconcepto* → tabla también predefinida que va a contener los subconceptos para cada concepto.
- *tabTopeConceptos* → tabla que contendrá los valores máximos de

- *tabTopesSubconceptos* → tabla que contendrá los valores máximos de gasto o ingreso por cada subconcepto para cada año.
- *tabGastos* → tabla principal que va a tener todos los gastos introducidos.

Todas las tablas van a tener un identificador autonumérico que será la clave de la tabla. En las tablas predefinidas no es necesario que sea autonumérico y basta con que sea numérico, ya que dichos valores se van a meter directamente en la base de datos.

La primera tabla que vamos a comentar es la de conceptos. En la tabla *tabConcepto* se va a guardar el texto del concepto y un campo que indique si el valor del campo va a ser positivo (es un ingreso) o negativo (es un gasto). La definición de esta tabla debe ser así:

tabConcepto		
	Field Name	Data Type
?	conId	Number
	conTexto	Text
	conPositivo	Yes/No

Los valores que se van a introducir en esta tabla son:

- Casa (negativo)
- Comida (negativo)
- Extra (negativo)
- Ingreso (positivo)
- Ocio (negativo)

La segunda tabla es la de los subconceptos. En la tabla *tabSubconcepto* se va a guardar el texto del subconcepto y el código del concepto con el que está relacionado. La definición debe ser así:

5. Proyectos complementos de programación

tabSubconcepto		
	Field Name	Data Type
scold	Number	
scoSubconcepto	Text	
scoConcepto	Number	

Los valores que se van a introducir en esta tabla son:

- Casa (1)
- Comida/Higiene (2)
- Farmacia (2)
- Regalos (3)
- Lotería (3)
- Niños (3)
- Transporte (3)
- Imagen/Sonido (3)
- Ropa (3)
- Varios (3)
- Internet/Teléfono (3)
- Nómina (4)
- Otros (4)
- Ocio (5)

En la tabla *tabTopeConceptos* se guardan los máximos anuales para cada concepto. Para ello, aparte de la clave, hacen falta tres campos, uno para el código del concepto, otro para el año y otro para la cantidad tope. El diseño de dicha tabla va a ser como el siguiente:

	Field Name	Data Type
	tcold	AutoNumber
	tcoConcepto	Number
	tcoAno	Number
	tcoTope	Number

Y en la tabla *tabTopeSubconceptos* se guardan los máximos anuales para cada subconcepto. En este caso hacen falta cuatro campos, aparte de la clave, uno para el código del concepto, otro para el código del subconcepto, otro para el año y otro para la cantidad tope. El diseño de dicha tabla va a ser como el siguiente:

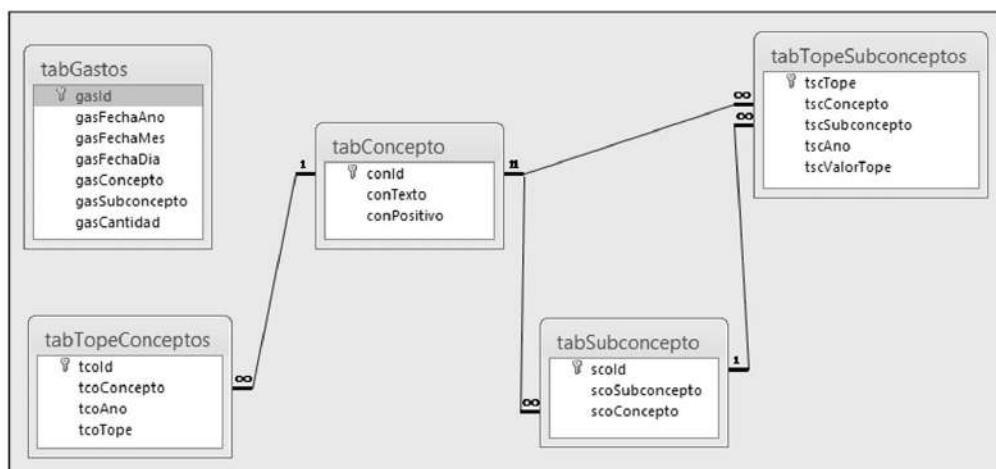
	Field Name	Data Type
	tscTope	AutoNumber
	tscConcepto	Number
	tscSubconcepto	Number
	tscAno	Number
	tscValorTope	Number

La última tabla es *tabGastos* y es la principal de todo el programa. En dicha tabla se van a guardar de forma separada todos los gastos e ingresos que se desean gestionar. Para ello es necesario introducir la fecha, el concepto, el subconcepto y la cantidad. En este caso se ha optado por separar la fecha en año, mes y día, teniendo tres campos separados. Esto se ha hecho así porque las fechas en Access no son muy estándares en cuanto a tratamiento a través de las consultas SQL y si se decidiera cambiar a otra base de datos diferente con un campo de tipo fecha, habría que realizar retoques en el código. De esta manera, no será necesario tocar nada. El diseño de esta tabla debe quedar así:

5. Proyectos complementos de programación

Field Name	Data Type
gasId	AutoNumber
gasFechaAno	Number
gasFechaMes	Number
gasFechaDia	Number
gasConcepto	Text
gasSubconcepto	Text
gasCantidad	Number

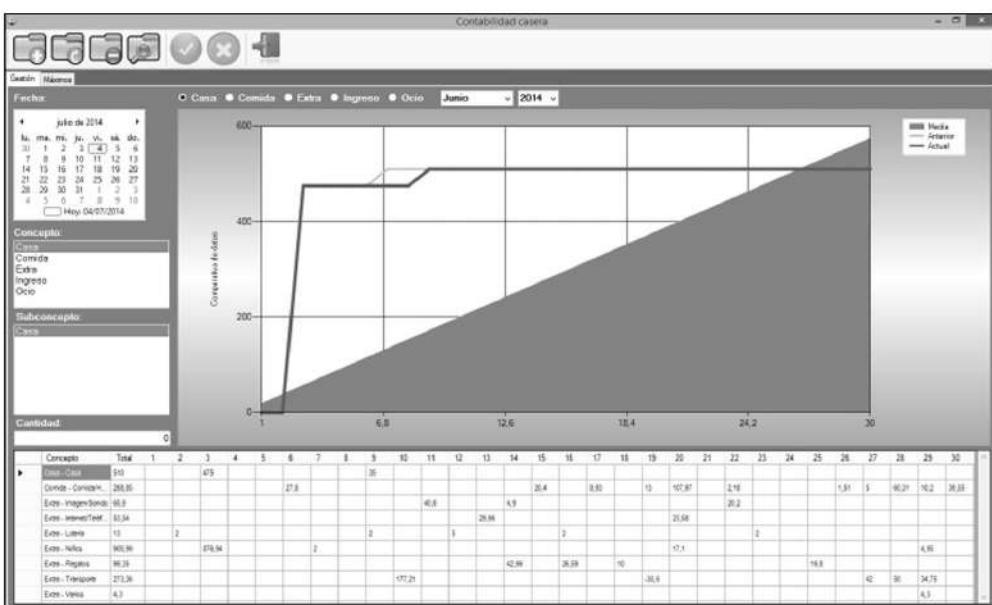
Hecho esto, las relaciones entre las tablas son las habituales entre los campos con las claves principales y las tablas que están relacionadas con ellas. En este caso, la salvedad es la tabla principal *tabGastos*, puesto que en ella se guardan directamente los campos concepto y subconcepto en formato texto, no sus códigos, así que dicha tabla no va a estar relacionada con las otras. La razón es que después se simplifican mucho las consultas cuando se usa dicha tabla, que es la más habitual. Lo que se pierde es optimización en los datos guardados, pero como tampoco son muy grandes, se ha optado por este diseño que gana en velocidad de ejecución. El diseño final de las relaciones queda de esta manera:



Una vez que se tengan las tablas ya definidas y los valores predefinidos introducidos, se comienza con el diseño de la aplicación. En este caso se ha decidido optar por un diseño de un solo formulario, en el cual va a haber varias partes diferenciadas. Se van a tener dos pestañas, una para la parte principal de gestión y otra para establecer los máximos que se modificarán año a año.

En la parte superior de la ventana se va a tener una barra de herramientas con los botones que van a controlar la parte de la gestión. También debe haber un botón para salir de la aplicación.

Lo mejor es ver el aspecto general que va a tener la ventana para cada pestaña y comentar cada apartado con más detenimiento. El diseño general para la ventana con la pestaña de gestión y que es el que se va a ver cuando arranque la aplicación es el de la siguiente imagen.



Esta pantalla, como se ha comentado, tiene varias partes bien diferenciadas. A continuación se van a explicar cada una de ellas.

5. Proyectos complementos de programación



En la parte superior, se tienen los botones de gestión de los ingresos y los gastos. Dichos botones, de izquierda a derecha, realizan las siguientes acciones:

- Añadir → añade un nuevo ingreso o gasto a la base de datos.
- Modificar → modifica un ingreso o gasto de la base de datos.
- Eliminar → elimina un ingreso o gasto de la base de datos.
- Consultar → consulta el importe de un ingreso o gasto.
- Aceptar → ejecuta la acción correspondiente con los datos que haya en ese momento en la ventana.
- Cancelar → cancela la acción que estaba pendiente.
- Salir → sale de la aplicación.





En la parte izquierda de la zona principal de la ventana, están los apartados correspondientes a un ingreso o a un gasto en la aplicación. Como se puede apreciar, los datos que se incluyen en cada elemento son la fecha, el concepto, el subconcepto y la cantidad. Siempre se va a introducir la cantidad en positivo, aunque después se pueda tener en cuenta que el concepto sea positivo o negativo.

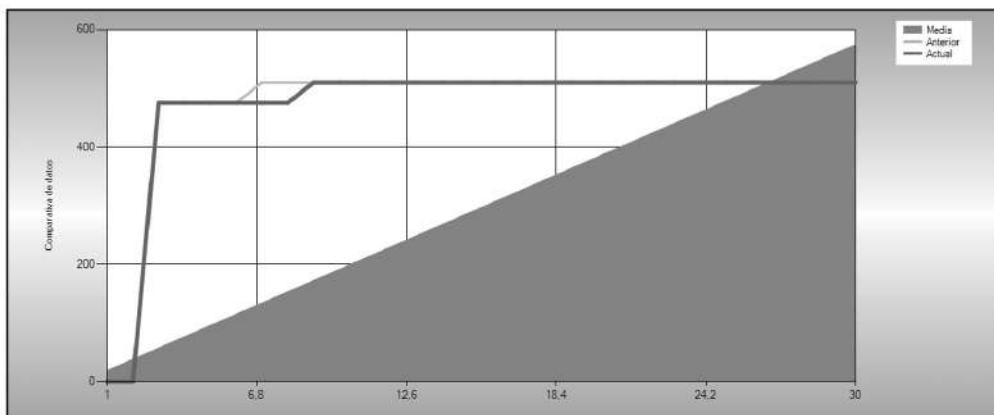


En la parte superior de la gráfica, se encuentran los selectores para visualizar los datos correspondientes. Por un lado, los botones de radio que permiten visualizar la gráfica para cada uno de los conceptos. En función

5. Proyectos complementos de programación

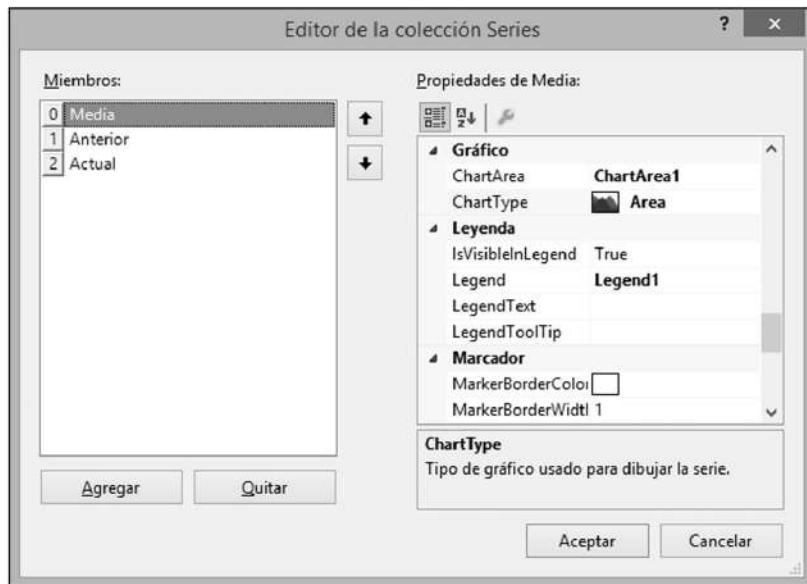
del botón de radio se podrán ver los datos, de izquierda a derecha, de los conceptos Casa, Comida, Extra, Ingreso y Ocio. Dado que estos datos están introducidos en la tabla correspondiente de forma predeterminada, no van a variar y por eso se pueden poner aquí cinco botones fijos.

En la parte de la derecha de la misma zona, se pueden escoger el mes y el año de visualización. En este caso no solamente van a hacer que la gráfica cambie de contenido, sino también la tabla que aparece en la parte inferior. Al modificar cualquiera de las dos cajas combinadas se van a actualizar automáticamente ambos elementos.



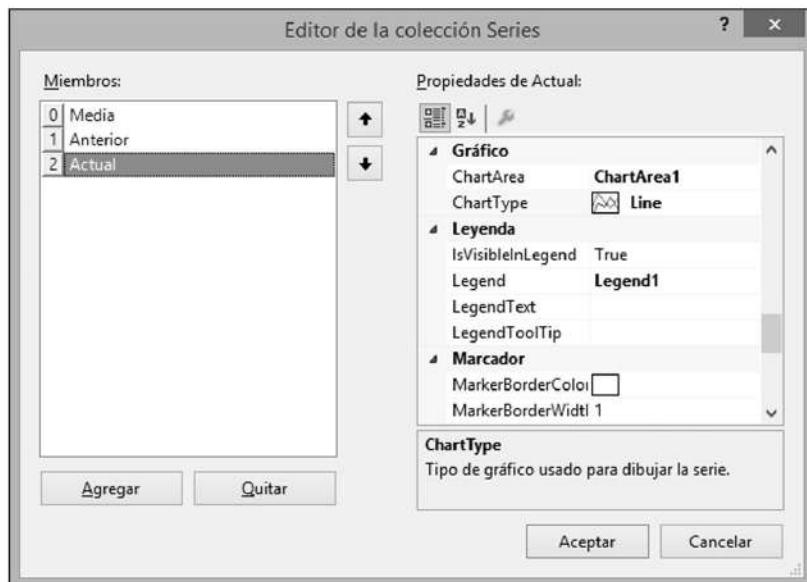
A la derecha de la zona principal se sitúa la gráfica. En el eje izquierdo se tienen las cantidades y en el eje inferior los días del mes. El contenido de la gráfica va a tener tres elementos. La parte que aparece como un área creciente es la media óptima para el mes en curso y el concepto seleccionado. Debe tener un color que permita contrastar bien con los otros dos. En este caso se ha optado por un azul más bien claro. El siguiente elemento es una línea clara que va a corresponder a los gastos o ingresos de ese concepto en el año anterior. De esta forma se puede ver la comparativa con los datos del año actual. En este caso se ha optado por una línea bastante fina de color amarillo oscuro. Por último, se tiene la línea con los datos del año actual. Se ha optado por una línea más gruesa que la anterior y en color rojo, para que sea vistosa.

Al realizar el diseño, la propiedad más importante de este elemento es la llamada *Series*, donde está la colección de series anteriormente descrita. Dentro de cada una, en la propiedad *ChartType* es donde se indica el tipo de gráfico que vamos a utilizar.



En el gráfico anterior, se aprecia que para la serie de las medias se ha utilizado el tipo de gráfico *Area*.

En los datos del año anterior y del actual se ha utilizado el tipo *Line* para dibujar los puntos, tal y como se aprecia en esta imagen:



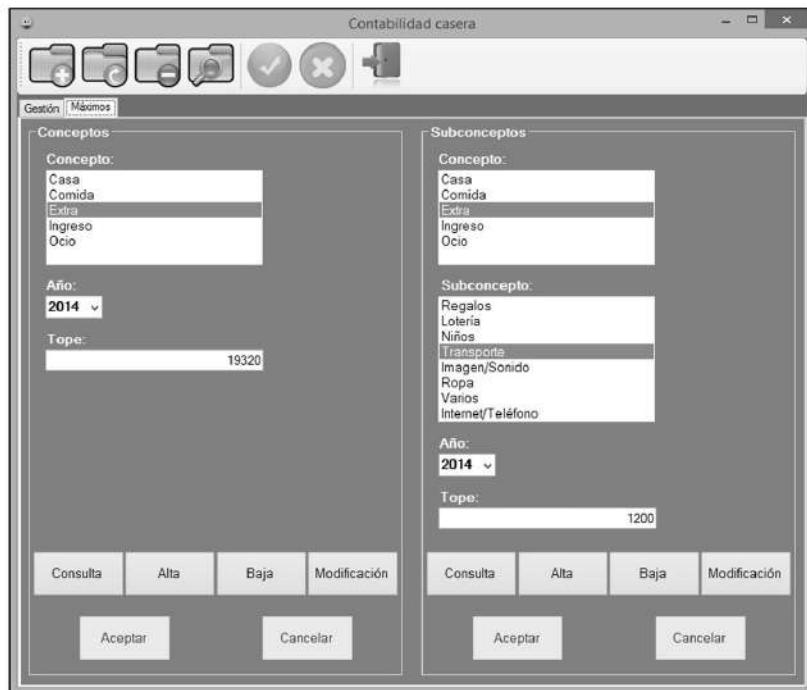
5. Proyectos complementos de programación

En la parte inferior se tiene una tabla con el resumen diario de cada uno de los conceptos y subconceptos, para que de un vistazo se pueda ver en detalle todo lo que se ha introducido para el mes correspondiente. Se puede ver un ejemplo en esta imagen:

Concepto	Total	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
Casa - Casa	910		675						35																							
Comida - ComidaH	288,85							27,8																								
Extra - Imagen/Sonido	65,9																40,8	4,8														
Extra - Internet/Teléf.	33,34																	29,96														
Extra - Lotería	12		2																													
Extra - Niños	906,99																															
Extra - Regalos	98,35																															
Extra - Transporte	273,38																															
Extra - Vehículo	4,3																															

La primera columna contiene el concepto y subconcepto, la segunda tiene la suma del mes y el resto de columnas corresponden a las cantidades diarias.

El diseño general para la ventana con la pestaña de máximos es el de la siguiente imagen.



En este caso hay dos partes bien diferenciadas que están agrupadas y separadas en la zona izquierda y la zona derecha de la pestaña.

Conceptos

Concepto:

- Casa
- Comida
- Extra**
- Ingreso
- Ocio

Año:

2014 ▾

Tope:

19320

Consulta **Alta** **Baja** **Modificación**

Aceptar **Cancelar**

5. Proyectos complementos de programación

Como se puede apreciar en la imagen anterior, en la parte izquierda se introducen el concepto, el año y el tope máximo para ese año.

Los botones que se usan para este apartado son los habituales:

- Consulta → consulta el tope máximo para el concepto y el año solicitados.
- Alta → añade un nuevo tope máximo para el concepto y el año indicados.
- Baja → elimina el tope máximo para el concepto y el año indicados.
- Modificación → modifica el tope máximo para el concepto y el año indicados.
- Aceptar → ejecuta la acción correspondiente con los datos que haya en ese momento en la ventana.
- Cancelar → cancela la acción que esté pendiente.

En la imagen de la página siguiente se puede apreciar que en la parte derecha se introduce el concepto, el subconcepto, el año y el tope máximo para ese año.

Los botones que se usan para este apartado son los habituales:

- Consulta → consulta el tope máximo para el subconcepto y el año solicitados.
- Alta → añade un nuevo tope máximo para el subconcepto y el año indicados.
- Baja → elimina el tope máximo para el subconcepto y el año indicados.
- Modificación → modifica el tope máximo para el subconcepto y el año indicados.
- Aceptar → ejecuta la acción correspondiente con los datos que haya en ese momento en la ventana.
- Cancelar → cancela la acción que esté pendiente.

Subconceptos

Concepto:

- Casa
- Comida
- Extra**
- Ingreso
- Ocio

Subconcepto:

- Regalos
- Lotería
- Niños
- Transporte**
- Imagen/Sonido
- Ropa
- Varios
- Internet/Teléfono

Año:

2014 ▾

Tope:

1200

Consulta Alta Baja Modificación

Aceptar **Cancelar**

5. Proyectos complementos de programación

A continuación se va a comentar el código usado en la aplicación. Lo primero es la definición de las variables generales que servirán para toda la aplicación. Van a ser necesarias variables para:

- Conectar con la base de datos.
- Controlar las tres acciones para los tres grupos de botones.
- Guardar cada elemento a la hora de las modificaciones.
- Guardar cada elemento a la hora de las eliminaciones.
- Realizar consultas a partir de la fila y la columna seleccionadas en la tabla inferior.

El código de creación e inicialización de dichas variables es:

```
// Variables principales
public OleDbConnection conBD = new OleDbConnection();
String sAcción = "";
String sAcciónConMax = "";
String sAcciónSubMax = "";
String sDiaModificar = "";
String sMesModificar = "";
String sAñoModificar = "";
String sConceptoModificar = "";
String sSubconceptoModificar = "";
String sCantidadModificar = "";
String sDiaEliminar = "";
String sMesEliminar = "";
String sAñoEliminar = "";
String sConceptoEliminar = "";
String sSubconceptoEliminar = "";
String sCantidadEliminar = "";
int iColumnaSeleccionada = 0;
int iFilaSeleccionada = 0;
```

Cuando arranca la aplicación hay que realizar una serie de acciones que permitan inicializar todos los elementos de pantalla con sus valores por defecto. La lista de acciones que se ejecutarán debe ser:

- Abrir la conexión con la base de datos.
- Activar y desactivar todos los botones correspondientes para poder usar sus acciones.

- Rellenar las listas de conceptos.
- Rellenar las listas de subconceptos.
- Rellenar las cajas combinadas con los años existentes en la base de datos.
- Inicializar las cajas de texto de los meses y los años con los valores en curso.
- Inicializar los botones de radio poniendo uno por defecto.
- Rellenar la gráfica con los datos del mes en curso.
- Rellenar la tabla de las cantidades mensuales para el mes en curso.

Estas acciones se van a realizar en el evento *Load* de carga inicial de la siguiente manera:

```
private void frmContabilidad_Load(object sender, EventArgs
e)
{
    // Se abre la conexión con la base de datos
    abrirConexion();
    // Se activan y desactivan los botones correspondientes
    activarAcciones();
    activarAccionesConceptos();
    activarAccionesSubconceptos();
    // Se muestran los conceptos en la lista
    leerConceptos();
    leerConceptosConMax();
    leerConceptosSubMax();
    // Se actualizan las cajas de mes y año con los actuales
    cboMeses.SelectedIndex = DateTime.Today.Month - 1;
    llenarAnos();
    llenarAnosConceptos();
    llenarAnosSubconceptos();
    // Se inicializa una de las cajas de opción
    radCasa.Checked = true;
    // Se inicializa el grid de datos
    llenarDatos(DateTime.Today.Month, DateTime.Today.Year);
}
```

A continuación se van a comentar cada una de las funciones de este arranque. Antes de nada, hay que tener en cuenta que por claridad en el

5. Proyectos complementos de programación

código, algunas veces las líneas están partidas y divididas en varias, pero en el original deben ir en la misma línea.

El inicio es la apertura de la conexión con la base de datos que, se hace de la manera habitual:

```
public void abrirConexion()
{
    // Se crea el string de conexión a la base de datos
    conBD.ConnectionString =
        @"Provider=Microsoft.ACE.OLEDB.12.0; Data Source= .\ \
        BD\Contabilidad.accdb; Persist Security Info=False";
    try
    {
        // Se abre la base de datos
        conBD.Open();
    }
    catch (Exception ex)
    {
        MessageBox.Show("Error de conexión" + ex);
    }
}
```

El código para activar los botones para el arranque es el que se puede ver en las siguientes funciones:

```
private void activarAcciones()
{
    butAnadir.Enabled = true;
    butEliminar.Enabled = true;
    butModificar.Enabled = true;
    butConsultar.Enabled = true;
    butAceptar.Enabled = false;
    butCancelar.Enabled = false;
    // Se cancela la acción que hay que realizar
    sAcción = "";
}

private void activarAccionesConceptos()
{
    butAltaConMax.Enabled = true;
    butBajaConMax.Enabled = true;
    butModificaciónConMax.Enabled = true;
    butConsultaConMax.Enabled = true;
    butAceptarConMax.Enabled = false;
    butCancelarConMax.Enabled = false;
```

```
// Se cancela la acción que hay que realizar
sAccionConMax = "";
}

private void activarAccionesSubconceptos()
{
    butAltaSubMax.Enabled = true;
    butBajaSubMax.Enabled = true;
    butModificacionSubMax.Enabled = true;
    butConsultaSubMax.Enabled = true;
    butAceptarSubMax.Enabled = false;
    butCancelarSubMax.Enabled = false;
    // Se cancela la acción que hay que realizar
    sAccionSubMax = "";
}
```

De la misma forma, para la ejecución de las acciones más adelante, el código para desactivar los botones es el que se puede ver en las siguientes funciones:

```
private void desactivarAcciones()
{
    butAnadir.Enabled = false;
    butEliminar.Enabled = false;
    butModificar.Enabled = false;
    butConsultar.Enabled = false;
    butAceptar.Enabled = true;
    butCancelar.Enabled = true;
}

private void desactivarAccionesConceptos()
{
    butAltaConMax.Enabled = false;
    butBajaConMax.Enabled = false;
    butModificacionConMax.Enabled = false;
    butConsultaConMax.Enabled = false;
    butAceptarConMax.Enabled = true;
    butCancelarConMax.Enabled = true;
}

private void desactivarAccionesSubconceptos()
{
    butAltaSubMax.Enabled = false;
    butBajaSubMax.Enabled = false;
    butModificacionSubMax.Enabled = false;
    butConsultaSubMax.Enabled = false;
    butAceptarSubMax.Enabled = true;
    butCancelarSubMax.Enabled = true;
}
```

5. Proyectos complementos de programación

Para llenar las listas de conceptos son necesarias otras tres funciones muy similares, cada una de las cuales va a llenar la lista correspondiente. El código para ello se puede ver a continuación:

```
private void leerConceptos()
{
    try
    {
        // Se vacía la lista
        lstConceptos.Items.Clear();
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM tabConcepto";
        OleDbCommand oleComando =
            new OleDbCommand(sConsulta, conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno
        while (oleReader.Read())
        {
            // Se añade la línea a la lista
            lstConceptos.Items.Add(Convert.ToString(
                oleReader["conTexto"]));
        }
        // Se deja marcado el primer concepto
        lstConceptos.SelectedIndex = 0;
    }
    catch (Exception ex)
    {
        // Si salta una excepción, se muestra el error
        MessageBox.Show("Error: " + ex);
    }
}
private void leerConceptosConMax()
{
    try
    {
        // Se vacía la lista
        lstConceptoMax.Items.Clear();
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM tabConcepto";
        OleDbCommand oleComando =
            new OleDbCommand(sConsulta, conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno
        while (oleReader.Read())
        {
            // Se añade la línea a la lista
            lstConceptoMax.Items.Add(Convert.ToString(
                oleReader["conTexto"]));
        }
        // Se deja marcado el primer concepto
        lstConceptoMax.SelectedIndex = 0;
    }
}
```

```
{  
    // Se añade la linea a la lista  
    lstConceptoMax.Items.Add(Convert.ToString(  
        oleReader["conTexto"]));  
}  
// Se deja marcado el primer concepto  
lstConceptoMax.SelectedIndex = 0;  
}  
catch (Exception ex)  
{  
    // Si salta una excepción, se muestra el error  
    MessageBox.Show("Error: " + ex);  
}  
}  
  
private void leerConceptosSubMax()  
{  
    try  
{  
        // Se vacía la lista  
        lstConceptoSubMax.Items.Clear();  
        // Se realiza la consulta  
        String sConsulta = "SELECT * "  
            + "FROM tabConcepto";  
        OleDbCommand oleComando =  
            new OleDbCommand(sConsulta, conBD);  
        OleDbDataReader oleReader =  
            oleComando.ExecuteReader();  
        // Se lee el resultado para ver si hay alguno  
        while (oleReader.Read())  
        {  
            // Se añade la linea a la lista  
            lstConceptoSubMax.Items.Add(Convert.ToString(  
                oleReader["conTexto"]));  
        }  
        // Se deja marcado el primer concepto  
        lstConceptoSubMax.SelectedIndex = 0;  
    }  
    catch (Exception ex)  
{  
        // Si salta una excepción, se muestra el error  
        MessageBox.Show("Error: " + ex);  
    }  
}
```

5. Proyectos complementos de programación

Hay que tener en cuenta que cuando se modifica un concepto, hay que modificar también las listas de subconceptos asociadas. Para ello se necesitan las funciones que obtienen las listas de subconceptos para el concepto que esté seleccionado. En este caso van a ser dos funciones, una para cada lista de subconceptos. Estas funciones van a tener este código:

```
private void leerSubconceptos()
{
    try
    {
        // Se vacía la lista
        lstSubconceptos.Items.Clear();
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM tabSubconcepto "
            + "WHERE scoConcepto="
            + (lstConceptos.SelectedIndex+1);
        OleDbCommand oleComando = new OleDbCommand(sConsulta,
            conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno
        while (oleReader.Read())
        {
            // Se añade la línea a la lista
            lstSubconceptos.Items.Add(Convert.ToString(
                oleReader["scoSubconcepto"]));
        }
        // Se deja marcado el primer subconcepto
        lstSubconceptos.SelectedIndex = 0;
    }
    catch (Exception ex)
    {
        // Si salta una excepción, se muestra el error
        MessageBox.Show("Error: " + ex);
    }
}

private void leerSubconceptosSubMax()
{
    try
    {
        // Se vacía la lista
        lstSubconceptoSubMax.Items.Clear();
        // Se realiza la consulta
```

```
String sConsulta = "SELECT * "
    + "FROM tabSubconcepto "
    + "WHERE scoConcepto="
    + (lstConceptoSubMax.SelectedIndex + 1);
OleDbCommand oleComando = new OleDbCommand(sConsulta,
    conBD);
OleDbDataReader oleReader =
    oleComando.ExecuteReader();
// Se lee el resultado para ver si hay alguno
while (oleReader.Read())
{
    // Se añade la linea a la lista
    lstSubconceptoSubMax.Items.Add(Convert.ToString(
        oleReader["scoSubconcepto"]));
}
// Se deja marcado el primer subconcepto
lstSubconceptoSubMax.SelectedIndex = 0;
}
catch (Exception ex)
{
    // Si salta una excepción, se muestra el error
    MessageBox.Show("Error: " + ex);
}
}
```

Para llenar las cajas combinadas de los años, en lugar de generar los años de forma fija, se va a utilizar el contenido de la propia base de datos para seleccionar aquellos años en los que haya gastos o ingresos. De esta forma, estar rellenas con un contenido coherente con la información que se tiene en la tabla. Las funciones para la obtención de los años y el relleno de todas las cajas combinadas tienen el siguiente código:

```
private void rellenarAnos()
{
    // Se vacía la lista
    cboAnos.Items.Clear();
    // Se realiza la consulta
    String sConsulta = "SELECT DISTINCT gasFechaAno "
        + "FROM tabGastos "
        + "ORDER BY gasFechaAno DESC";
    OleDbCommand oleComando = new OleDbCommand(sConsulta,
        conBD);
    OleDbDataReader oleReader = oleComando.ExecuteReader();
    // Se lee el resultado para ver si hay alguno
    while (oleReader.Read())
    {
        // Se añade la linea a la lista
    }
}
```

5. Proyectos complementos de programación

```
        cboAnos.Items.Add(Convert.ToString(
            oleReader["gasFechaAño"]));
    }
    // Se deja marcado el primer concepto
    cboAnos.SelectedIndex = 0;
}

private void rellenarAnosConceptos()
{
    // Se vacía la lista
    cboAnoMax.Items.Clear();
    // Se realiza la consulta
    String sConsulta = "SELECT DISTINCT gasFechaAño "
        + "FROM tabGastos "
        + "ORDER BY gasFechaAño DESC";
    OleDbCommand oleComando = new OleDbCommand(sConsulta,
        conBD);
    OleDbDataReader oleReader = oleComando.ExecuteReader();
    // Se lee el resultado para ver si hay alguno
    while (oleReader.Read())
    {
        // Se añade la línea a la lista
        cboAnoMax.Items.Add(Convert.ToString(
            oleReader["gasFechaAño"]));
    }
    // Se deja marcado el primer concepto
    cboAnoMax.SelectedIndex = 0;
}

private void rellenarAnosSubconceptos()
{
    // Se vacía la lista
    cboAnoSubMax.Items.Clear();
    // Se realiza la consulta
    String sConsulta = "SELECT DISTINCT gasFechaAño "
        + "FROM tabGastos "
        + "ORDER BY gasFechaAño DESC";
    OleDbCommand oleComando = new OleDbCommand(sConsulta,
        conBD);
    OleDbDataReader oleReader = oleComando.ExecuteReader();
    // Se lee el resultado para ver si hay alguno
    while (oleReader.Read())
    {
        // Se añade la línea a la lista
        cboAnoSubMax.Items.Add(Convert.ToString(
            oleReader["gasFechaAño"]));
    }
    // Se deja marcado el primer concepto
    cboAnoSubMax.SelectedIndex = 0;
}
```

Cuando se modifica el año en una de las cajas combinadas, de la misma forma que cuando se modifica el mes, se actualiza la gráfica. La función que dibuja la gráfica es una de las más complicadas, puesto que tiene que obtener bastante información para cada una de las series. Las acciones que debe ejecutar esta función son las siguientes:

- Obtener el número de días del mes en curso.
- Poner el texto general para la gráfica.
- Inicializar los puntos para todas las series.
- Establecer los límites del eje horizontal.
- Inicializar los datos para cada *array* de cada serie.
- Obtener de la base de datos los datos del mismo mes del año anterior y guardarlos en el *array*.
- Obtener de la base de datos los datos del mes en curso y guardarlos en el *array*.
- Obtener el máximo del mes en curso de la base de datos y calcular la media para cada día.
- Acumular los datos obtenidos, puesto que se quiere que la gráfica sea acumulativa.
- Asignar todos los puntos a la gráfica.

El código para realizar todas estas acciones es el que se puede ver a continuación en esta función:

```
private void dibujarGrafica(String sConcepto, int iMes,
    int iAno)
{
    // Se rellena la tabla con los datos de la tabla
    // correspondiente
    int iDiasMes = diasMes(iMes, iAno);
    // Los datos que se van a guardar en cada uno de los
    // índices son los siguientes
    // - 0: días del mes
    // - 1: media de los datos para el mes actual
    // - 2: datos del mes del año anterior
    // - 3: datos del mes actual
    // ...
```

5. Proyectos complementos de programación

```
graDatos.ChartAreas[0].AxisY.Title =
    "Comparativa de datos";
// Se vacian las series
graDatos.Series[0].Points.Clear(); // medias
graDatos.Series[1].Points.Clear(); // anteriores
graDatos.Series[2].Points.Clear(); // actuales
// Se establecen los límites del eje horizontal
graDatos.ChartAreas[0].AxisX.Minimum = 1;
graDatos.ChartAreas[0].AxisX.Maximum = iDiasMes;
// Se crean los valores para los diferentes elementos
double[,] elementosMes = new double[iDiasMes, 3];
// Se llenan los valores por defecto para cada serie
for (int i = 1; i <= iDiasMes; i++)
{
    // Se crean valores iniciales
    elementosMes[i - 1, 0] = 0;
    elementosMes[i - 1, 1] = 0;
    elementosMes[i - 1, 2] = 0;
}
try
{
    // Se obtienen los datos del año anterior
    String sConsulta = "Select gasFechaDia,
        SUM(gasCantidad) As Suma "
        + "From tabGastos "
        + "Where gasFechaMes=" + iMes + " "
        + "And gasFechaAno=" + (iAno - 1) + " "
        + "And gasConcepto=''" + sConcepto + "' "
        + "Group by gasFechaDia "
        + "Order by gasFechaDia";
    OleDbCommand oleComando = new OleDbCommand(sConsulta,
        conBD);
    OleDbDataReader oleReader =
        oleComando.ExecuteReader();
    // Se lee el resultado para ver si hay alguno
    while (oleReader.Read())
    {
        // Se pone la cantidad
        elementosMes[Convert.ToInt16(
            oleReader["gasFechaDia"])-1, 1] =
            Convert.ToDouble(oleReader["Suma"]);
    }
    // Se obtienen los datos del año actual
    sConsulta = "Select gasFechaDia,
        SUM(gasCantidad) As Suma "
        + "From tabGastos "
        + "Where gasFechaMes=" + iMes + " "
        + "And gasFechaAno=" + iAno + " "
        + "And gasConcepto=''" + sConcepto + "' "
```

```
+ "Order by gasFechaDia";
oleComando = new OleDbCommand(sConsulta, conBD);
oleReader = oleComando.ExecuteReader();
// Se lee el resultado para ver si hay alguno
while (oleReader.Read())
{
    // Se pone la cantidad
    elementosMes[Convert.ToInt16(
        oleReader["gasFechaDia"])-1, 2] =
        Convert.ToDouble(oleReader["Suma"]);
}
// Se obtienen los datos del tope para el año actual
sConsulta = "Select tcoTope "
+ "From tabTopeConceptos, tabConcepto "
+ "Where tcoAno=" + iAno + " "
+ "And tcoConcepto=conId "
+ "And contExito=''" + sConcepto + "'";
oleComando = new OleDbCommand(sConsulta, conBD);
oleReader = oleComando.ExecuteReader();
double topeMes = 0;
// Se lee el resultado para ver si hay alguno
if (oleReader.Read())
{
    // Se obtiene el tope
    topeMes = Convert.ToInt32(oleReader["tcoTope"])
        / 12;
}
// Se rellenan los datos con el tope del mes
for (int i = 1; i <= iDiasMes; i++)
{
    elementosMes[i-1, 0] = topeMes / iDiasMes;
}
// Se acumulan los datos y se rellenan todos los que
// faltan
for (int i = 2; i <= iDiasMes; i++)
{
    elementosMes[i - 1, 0] = elementosMes[i - 2, 0]
        + elementosMes[i - 1, 0];
    elementosMes[i - 1, 1] = elementosMes[i - 2, 1]
        + elementosMes[i - 1, 1];
    elementosMes[i - 1, 2] = elementosMes[i - 2, 2]
        + elementosMes[i - 1, 2];
}
// Se asignan los puntos a la gráfica
```

5. Proyectos complementos de programación

```
for (int i = 1; i <= iDiasMes; i++)
{
    // Se crean valores iniciales
    DataPoint p0 = new DataPoint(i,
        elementosMes[i - 1, 0]);
    DataPoint p1 = new DataPoint(i,
        elementosMes[i - 1, 1]);
    DataPoint p2 = new DataPoint(i,
        elementosMes[i - 1, 2]);
    graDatos.Series[0].Points.Add(p0);
    graDatos.Series[1].Points.Add(p1);
    graDatos.Series[2].Points.Add(p2);
}
catch (Exception ex)
{
    MessageBox.Show("Error: " + ex.Message);
}
}
```

La otra función para el arranque con bastante complejidad es la que va a llenar la tabla de la parte inferior. En esta tabla se van colocando los gastos e ingresos día a día para el mes en curso con una línea para cada subconcepto. Las acciones que se deben ejecutar son:

- Primero se calculan los días del mes.
- Se inicializa la tabla con una línea y el número de columnas necesarias.
- Se asigna un ancho diferente a las dos columnas iniciales de la tabla.
- Se calcula y asigna el ancho para el resto de columnas en función de los días del mes.
- Se ponen los títulos de las columnas.
- Se obtienen los ingresos y gastos del mes en curso.
- Se calculan los acumulados para cada día por subconcepto.
- Se rellena cada fila de la tabla.
- Se calculan y colocan los totales por línea de subconcepto.

Para realizar todas estas acciones, el código fuente de esta función es el que se muestra a continuación:

```
private void rellenarDatos(int iMes, int iAno)
{
    // Se rellenan los datos del mes actual
    int iDiasMes = diasMes(iMes, iAno);
    // Se inicializa el grid con una sola línea
    grdDatos.RowCount = 0;
    // Se ponen dos columnas más que el número de días que
    // tenga el mes
    grdDatos.ColumnCount = iDiasMes + 2;
    // Se cambia el ancho de la columna inicial
    grdDatos.Columns[0].Width = 100;
    // Se cambia el ancho de la columna inicial de datos
    grdDatos.Columns[1].Width = 50;
    // Se pone primero el título de la primera columna de
    // datos
    grdDatos.Columns[0].HeaderText = "Concepto";
    grdDatos.Columns[1].HeaderText = "Total";
    // Se llenan primero los títulos de las columnas con
    los
    // días del mes
    for (int iCont=1; iCont<=iDiasMes; iCont++)
    {
        // Se asigna a cada elemento del título el día
        // correspondiente
        grdDatos.Columns[iCont+1].HeaderText =
            iCont.ToString();
        // Se cambia el ancho de las columnas
        grdDatos.Columns[iCont + 1].Width =
            (grdDatos.Width-200)/iDiasMes;
    }
    try
    {
        // A continuación se obtienen el resto de datos para
        // llenar el Grid
        // Se obtienen los datos del mes actual
        String sConsulta = "Select gasConcepto,
                            gasSubconcepto, gasFechaDia, "
                            + "SUM(gasCantidad) As Suma "
                            + "From tabGastos "
                            + "Where gasFechaMes=" + iMes + " "
                            + "And gasFechaAno=" + iAno + " "
                            + "Group by gasConcepto, gasSubconcepto,
                            gasFechaDia "
                            + "Order by gasConcepto, gasSubconcepto,
                            gasFechaDia";
```

5. Proyectos complementos de programación

```
        conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        String sSubconcepto = "";
        // Se lee el resultado para ver si hay alguno
        while (oleReader.Read())
        {
            // Primero se mira a ver si el subconcepto es
            // diferente
            if (sSubconcepto != Convert.ToString(
                oleReader["gasConcepto"]) + " - "
                + Convert.ToString(oleReader["gasSubconcepto"]))
            {
                sSubconcepto =
                    Convert.ToString(oleReader["gasConcepto"])
                    + " - " + Convert.ToString(
                        oleReader["gasSubconcepto"]);
                grdDatos.RowCount = grdDatos.RowCount + 1;
            }
            // Se pone el texto en la primera columna de la
            // fila
            grdDatos.Rows[grdDatos.RowCount - 1].Cells[0]
                .Value = sSubconcepto;
            // Se pone la cantidad en el día correspondiente
            grdDatos.Rows[grdDatos.RowCount - 1].Cells[
                Convert.ToInt16(oleReader["gasFechaDia"])]
                .Value =
                    Convert.ToDouble(oleReader["Suma"]);
            // Se suma la cantidad en la primera columna del
            // total
            double dTotal = 0;
            if (grdDatos.Rows[grdDatos.RowCount -
                1].Cells[1].Value==""))
            {
                dTotal = 0;
            }
            else
            {
                dTotal = Convert.ToDouble(grdDatos.Rows[
                    grdDatos.RowCount - 1].Cells[1].Value);
            }
            grdDatos.Rows[grdDatos.RowCount -
                1].Cells[1].Value = dTotal
                + Convert.ToDouble(oleReader["Suma"]);
        }
    }
    catch (Exception ex)
```

```
        MessageBox.Show("Error: " + ex.Message);  
    }  
}
```

Una vez que se tiene todo inicializado, lo siguiente que hay que hacer es programar los eventos de cada elemento de la ventana. Empezando por la parte principal, lo que deben hacer los eventos para los botones de gestión es marcar la acción que se debe hacer y desactivarse, para después activar los botones de *Aceptar* y *Cancelar*. Pero también hay algo más en el caso de los botones de *Modificación* y *Baja*. Hay que guardar los datos que se tienen en pantalla en el momento en que se pulsan esos botones. Dichos datos serán posteriormente utilizados para realizar la acción correspondiente. El código para los eventos de estos cuatro botones es el siguiente:

```
private void butAnadir_Click(object sender, EventArgs e)  
{  
    // Se indica la acción que hay que realizar  
    sAcción = "Alta";  
    // Se activan y desactivan los botones correspondientes  
    desactivarAcciones();  
}  
  
private void butModificar_Click(object sender, EventArgs e)  
{  
    // Se indica la acción que hay que realizar  
    sAcción = "Modificación";  
    // Se activan y desactivan los botones correspondientes  
    desactivarAcciones();  
    // Se guardan los datos que hay que modificar  
    sDiaModificar =  
        calFecha.SelectionRange.Start.Day.ToString();  
    sMesModificar =  
        calFecha.SelectionRange.Start.Month.ToString();  
    sAñoModificar =  
        calFecha.SelectionRange.Start.Year.ToString();  
    sConceptoModificar = lstConceptos.SelectedItem.ToString();  
    sSubconceptoModificar =  
        lstSubconceptos.SelectedItem.ToString();  
    sCantidadModificar = txtCantidad.Text;  
}  
  
private void butEliminar_Click(object sender, EventArgs e)  
{  
    // Se indica la acción que hay que realizar  
    sAcción = "Baja";
```

5. Proyectos complementos de programación

```
// Se activan y desactivan los botones correspondientes  
desactivarAcciones();  
// Se guardan los datos que hay que eliminar  
sDiaEliminar =  
    calFecha.SelectionRange.Start.Day.ToString();  
sMesEliminar =  
    calFecha.SelectionRange.Start.Month.ToString();  
sAnoEliminar =  
    calFecha.SelectionRange.Start.Year.ToString();  
sConceptoEliminar = lstConceptos.SelectedItem.ToString();  
sSubconceptoEliminar =  
    lstSubconceptos.SelectedItem.ToString();  
sCantidadEliminar = txtCantidad.Text;  
}  
private void butConsultar_Click(object sender, EventArgs e)  
{  
    // Se indica la acción que hay que realizar  
    sAcción = "Consulta";  
    // Se activan y desactivan los botones correspondientes  
    desactivarAcciones();  
}
```

Los siguientes botones que se van a comentar son los de aceptar y cancelar las acciones que se han designado. Por supuesto, el de cancelación es muy sencillo y lo único que debe hacer es vaciar la posible acción pendiente y volver a activar los botones principales. El código debe ser:

```
private void butCancelar_Click(object sender, EventArgs e)  
{  
    // Se termina la acción realizada  
    sAcción = "";  
    // Se activan y desactivan los botones correspondientes  
    activarAcciones();  
}
```

El botón para aceptar la acción es el que tiene mayor complejidad, por lo que se requiere una explicación más exhaustiva de lo que tiene que hacer. A continuación, se explican en una lista cada una de las acciones que se deben ejecutar en el evento de pulsación del botón:

- Se obtienen los datos de los diferentes elementos de la ventana.
- Si se trata de un alta:
 - Se obtiene la cantidad y se convierte a número doble.

- Si se trata de una modificación:
 - Se ejecuta la consulta de eliminación de los datos anteriores.
 - Se obtiene la cantidad y se convierte a número doble.
 - Se ejecuta la consulta de inserción.
- Si se trata de una baja:
 - Se ejecuta la consulta de eliminación.
- Si se trata de una consulta:
 - Se verifica que haya algún elemento válido seleccionado en la tabla de ingresos y gastos.
 - Se obtienen los datos para realizar la búsqueda.
 - Se ejecuta la consulta de búsqueda del elemento seleccionado.
 - Se colocan los resultados de la búsqueda en los elementos de la ventana.
- Para todos los casos excepto la consulta, se actualizan la gráfica y la tabla con los elementos actualizados. También se vacían los elementos de la pantalla.
- Para todos los casos, se vacía la acción pendiente y se activan los botones principales de nuevo.

Y el código que se corresponde con todas estas acciones es el que se muestra a continuación:

```
private void butAceptar_Click(object sender, EventArgs e)
{
    // Se obtienen los datos
    String sDia =
        calFecha.SelectionRange.Start.Day.ToString();
    String sMes =
        calFecha.SelectionRange.Start.Month.ToString();
    String sAño =
        calFecha.SelectionRange.Start.Year.ToString();
```

5. Proyectos complementos de programación

```
String sSubconcepto =
    1stSubconceptos.SelectedItem.ToString();
String sCantidad = "0";
// Se chequea la acción que hay que ejecutar
if (sAcción == "Alta")
{
    try
    {
        sCantidad = Convert.ToDouble(
            txtCantidad.Text.Replace(".", " , ")).ToString()
            .Replace(" , ", ".");
    }
    catch (Exception ex)
    {
        MessageBox.Show("Error: " + ex.Message);
    }
    // Se ejecuta la consulta de inserción
    String sConsulta = "INSERT INTO tabGastos "
        + "(gasFechaAno, gasFechaMes, gasFechaDia,
        gasConcepto,
        + "gasSubconcepto, gasCantidad) "
        + "VALUES (" + sAno + ", " + sMes + ", " + sDia
        + ", " + sConcepto + ", "
        + sSubconcepto + ", " + sCantidad + ")";
    OleDbCommand oleComando = new OleDbCommand();
    oleComando.CommandType = CommandType.Text;
    oleComando.CommandText = sConsulta;
    oleComando.Connection = conBD;
    // Se ejecuta el comando de inserción
    oleComando.ExecuteNonQuery();
}
else
{
    if (sAcción == "Modificación")
    {
        // Primero se elimina el elemento anterior
        String sConsulta = "DELETE FROM tabGastos "
            + "WHERE gasFechaAno= " + sAnoModificar + " "
            + "AND gasFechaMes= " + sMesModificar + " "
            + "AND gasFechaDia= " + sDiaModificar + " "
            + "AND gasConcepto= " + sConceptoModificar
            + " "
            + "AND gasSubconcepto= "
            + sSubconceptoModificar + " "
            + "AND gasCantidad= " + sCantidadModificar;
        OleDbCommand oleComando = new OleDbCommand();
        oleComando.CommandType = CommandType.Text;
        oleComando.CommandText = sConsulta;
        oleComando.Connection = conBD;
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
oleComando.ExecuteNonQuery();
// Despues se añade el nuevo elemento
try
{
    sCantidad = Convert.ToDouble(
        txtCantidad.Text.Replace(".", ",").Replace(",","."));
    .ToString().Replace(".", ",").Replace(",","."));
}
catch (Exception ex)
{
    MessageBox.Show("Error: " + ex.Message);
}
// Se ejecuta la consulta de insercion
sConsulta = "INSERT INTO tabGastos "
+ "(gasFechaAno, gasFechaMes, gasFechaDia,
gasConcepto, "
+ "gasSubconcepto, gasCantidad) "
+ "VALUES (" + sAno + ", " + sMes + ", "
+ sDia + ", " + sConcepto + ", "
+ sSubconcepto + ", " + sCantidad + ")";
oleComando = new OleDbCommand();
oleComando.CommandType = CommandType.Text;
oleComando.CommandText = sConsulta;
oleComando.Connection = conBD;
// Se ejecuta el comando de insercion
oleComando.ExecuteNonQuery();
}
else
{
    if (sAccion == "Baja")
    {
        // Se elimina el elemento
        String sConsulta = "DELETE FROM tabGastos "
        + "WHERE gasFechaAno= " + sAnoEliminar
        + " "
        + "AND gasFechaMes= " + sMesEliminar + " "
        + "AND gasFechaDia= " + sDiaEliminar + " "
        + "AND gasConcepto= " + sConceptoEliminar
        + " "
        + "AND gasSubconcepto= "
        + sSubconceptoEliminar + " "
        + "AND gasCantidad= "
        + sCantidadEliminar.Replace(".", ",").Replace(",","."));
        OleDbCommand oleComando = new OleDbCommand();
        oleComando.CommandType = CommandType.Text;
        oleComando.CommandText = sConsulta;
        oleComando.Connection = conBD;
        // Se ejecuta el comando de eliminacion
        oleComando.ExecuteNonQuery();
    }
}
```

5. Proyectos complementos de programación

```
else
{
    // Se chequea si hay seleccionado un día o no
    if (iColumnaSeleccionada>0)
    {
        String sConceptoSeleccionado =
            grdDatos.Rows[iFilaSeleccionada]
            .Cells[0].Value.ToString();
        String sSubconceptoSeleccionado =
            sConceptoSeleccionado.Substring(
                sConceptoSeleccionado.IndexOf("-"
            )
            + 3);
        sConceptoSeleccionado =
            sConceptoSeleccionado.Sub-
            string(0,
                sConceptoSeleccionado.IndexOf(
                    " - "));
        // Se consulta el elemento seleccionado en
        // la lista
        String sConsulta = "Select gasCantidad "
            +
            "From tabGastos "
            +
            "Where gasFechaDia= "
            +
            iColumnaSeleccionada + " "
            +
            "And gasFechaMes= "
            +
            (cboMeses.SelectedIndex + 1) + " "
            +
            "And gasFechaAño= "
            +
            Convert.ToInt16(cboAnos.Text) + "
            "
            +
            "And gasConcepto= '"
            +
            sConceptoSeleccionado + "'"
            +
            "And gasSubconcepto= '"
            +
            sSubconceptoSeleccionado + "'";
        OleDbCommand oleComando =
            new OleDbCommand(sConsulta,
            conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay
        // alguno
        while (oleReader.Read())
        {
            // Se ponen los datos en pantalla
            calFecha.SetDate(new DateTime(
                Convert.ToInt16(cboAnos.Text),
                (cboMeses.SelectedIndex + 1),
                iColumnaSeleccionada));
            lstConceptos.SelectedItem =
                sConceptoSeleccionado;
```

```
    sSubconceptoSeleccionado;
    txtCantidad.Text =
        Convert.ToString(oleReader[
            "gasCantidad"]).Replace(",",
            "."));
    }
}
}
}
if (sAccion != "Consulta")
{
    // Se actualiza la gráfica
    dibujarGrafica(sConcepto, Convert.ToInt16(sMes),
        Convert.ToInt16(sAno));
    // Se actualiza el grid de datos
    llenarDatos(Convert.ToInt16(sMes),
        Convert.ToInt16(sAno));
    // Se vacían los elementos de pantalla
    calFecha.SelectionStart = calFecha.TodayDate;
    lstConceptos.SelectedIndex = 0;
    txtCantidad.Text = "0";
}
// Se termina la acción realizada
sAccion = "";
// Se activan y desactivan los botones correspondientes
activarAcciones();
}
```

Para realizar las funciones anteriores, se ha utilizado una función auxiliar para obtener los días de un mes. A dicha función hay que pasarle como parámetros el mes deseado y también el año, puesto que se debe tener en cuenta si puede ser un año bisiesto o no. En este caso, para ser bisiesto simplemente se calcula si es múltiplo de 4 y no es múltiplo de 100. Teniendo en cuenta que los años van a ser de este siglo o como mucho del pasado si se tuvieran datos anteriores, no es necesario complicarlo más. El código fuente de esta función es el siguiente:

```
private int diasMes(int _mes, int _ano)
{
    // Devuelve el número de días que tiene el mes
    int bDiasMes = 0;
    switch (_mes)
    {
        case 1:
            bDiasMes = 31;
```

5. Proyectos complementos de programación

```
case 2:  
    bDiasMes = 28;  
    // Se mira a ver si el año es bisiesto  
    if (_ano%4 == 0) && (_ano%100 != 0)  
    {  
        bDiasMes = 29;  
    }  
    break;  
case 3:  
    bDiasMes = 31;  
    break;  
case 4:  
    bDiasMes = 30;  
    break;  
case 5:  
    bDiasMes = 31;  
    break;  
case 6:  
    bDiasMes = 30;  
    break;  
case 7:  
    bDiasMes = 31;  
    break;  
case 8:  
    bDiasMes = 31;  
    break;  
case 9:  
    bDiasMes = 30;  
    break;  
case 10:  
    bDiasMes = 31;  
    break;  
case 11:  
    bDiasMes = 30;  
    break;  
case 12:  
    bDiasMes = 31;  
    break;  
}  
return bDiasMes;  
}
```

También se ha comentado que cuando se modifica la selección de uno de los conceptos, automáticamente se deben llenar los subconceptos. Esto se hace de manera sencilla utilizando el evento que indica que se ha cambiado el índice de selección, tal y como se puede ver en esta función:

```
private void lstConceptos_SelectedIndexChanged(object sender,  
EventArgs e)
```

```
{  
    // Se actualiza la lista de subconceptos en función del  
    // concepto elegido  
    leerSubconceptos();  
}
```

Otros eventos que se han comentado son los de los botones de radio. Cuando se selecciona uno diferente, se tiene que actualizar la gráfica. Para ello lo que hay que hacer es chequear si ha cambiado el valor en cada uno de ellos y, si está seleccionado, entonces dibujar la gráfica de dicho concepto. El código de estos eventos es el que sigue:

```
private void radCasa_CheckedChanged(object sender,  
    EventArgs e)  
{  
    if (radCasa.Checked)  
    {  
        // Se rellena la gráfica  
        dibujarGrafica("Casa", cboMeses.SelectedIndex + 1,  
            Convert.ToInt16(cboAnos.Text));  
    }  
}  
  
private void radComida_CheckedChanged(object sender,  
    EventArgs e)  
{  
    if (radComida.Checked)  
    {  
        // Se rellena la gráfica  
        dibujarGrafica("Comida", cboMeses.SelectedIndex + 1,  
            Convert.ToInt16(cboAnos.Text));  
    }  
}  
  
private void radExtra_CheckedChanged(object sender,  
    EventArgs e)  
{  
    if (radExtra.Checked)  
    {  
        // Se rellena la gráfica  
        dibujarGrafica("Extra", cboMeses.SelectedIndex + 1,  
            Convert.ToInt16(cboAnos.Text));  
    }  
}
```

5. Proyectos complementos de programación

```
private void radIngreso_CheckedChanged(object sender,
EventArgs e)
{
    if (radIngreso.Checked)
    {
        // Se rellena la gráfica
        dibujarGrafica("Ingreso", cboMeses.SelectedIndex + 1,
Convert.ToInt16(cboAnos.Text));
    }
}

private void radOcio_CheckedChanged(object sender,
EventArgs e)
{
    if (radOcio.Checked)
    {
        // Se rellena la gráfica
        dibujarGrafica("Ocio", cboMeses.SelectedIndex + 1,
Convert.ToInt16(cboAnos.Text));
    }
}
```

En caso de que se cambien tanto el mes como el año, además de realizar una actualización de la gráfica, se deberán actualizar los datos de la tabla. En el caso de la gráfica, se tendrá que chequear el concepto que se tiene que actualizar. Como el código debe ser el mismo, lo más sencillo es llamar en uno de los elementos al otro. El resultado será como sigue:

```
private void cboMeses_SelectedIndexChanged(object sender,
EventArgs e)
{
    if (radCasa.Checked)
    {
        // Se rellena la gráfica
        dibujarGrafica("Casa", cboMeses.SelectedIndex + 1,
Convert.ToInt16(cboAnos.Text));
    }
    else
    {
        if (radComida.Checked)
        {
            // Se rellena la gráfica
            dibujarGrafica("Comida", cboMeses.SelectedIndex
+ 1, Convert.ToInt16(cboAnos.Text));
        }
    }
}
```

```
if (radExtra.Checked)
{
    // Se rellena la gráfica
    dibujarGrafica("Extra", cboMeses.SelectedIndex
        + 1, Convert.ToInt16(cboAnos.Text));
}
else
{
    if (radIngreso.Checked)
    {
        // Se rellena la gráfica
        dibujarGrafica("Ingreso",
            cboMeses.SelectedIndex + 1,
            Convert.ToInt16(cboAnos.Text));
    }
    else
    {
        if (radOcio.Checked)
        {
            // Se rellena la gráfica
            dibujarGrafica("Ocio",
                cboMeses.SelectedIndex + 1,
                Convert.ToInt16(cboAnos.Text));
        }
    }
}
// Se inicializa el grid de datos
if (cboAnos.Text.Length>0)
{
    llenarDatos(cboMeses.SelectedIndex + 1,
        Convert.ToInt16(cboAnos.Text));
}

private void cboAnos_SelectedIndexChanged(object sender,
    EventArgs e)
{
    cboMeses_SelectedIndexChanged(sender, e);
}
```

Otro evento que se debe controlar en la parte principal es la pulsación en una de las celdas de la tabla de datos de la parte inferior. Para cuando se hacen las consultas, hay que saber qué posición de esta tabla se ha seleccionado, con lo que lo más sencillo es guardar dichas posiciones

5. Proyectos complementos de programación

cada vez que sean modificadas. Este evento es sencillo y queda como se muestra aquí:

```
private void grdDatos_CellClick(object sender,
    DataGridViewCellEventArgs e)
{
    // Se guarda la celda seleccionada
    iColumnaSeleccionada = e.ColumnIndex - 1;
    iFilaSeleccionada = e.RowIndex;
}
```

La siguiente parte del código es la correspondiente a la pestaña *Máximos*. Como ya se ha comentado anteriormente, esta pestaña tiene dos partes muy diferenciadas y de la misma forma, el código está diferenciado para cada una de las partes. La parte de la inicialización de los elementos está realizada en conjunto con el resto de inicializaciones y desde el evento *Load* de la ventana ya se queda todo precargado.

En la zona izquierda de la ventana, dentro del grupo *Conceptos*, se tienen los cuatro botones habituales para la gestión. En cada uno de los botones, de la misma forma que siempre, se va a marcar la acción que se quiere ejecutar y también se van a deshabilitar dichos botones para habilitar los de *Aceptar* y *Cancelar*. El código para estos cuatro botones principales será así:

```
private void butConsultaConMax_Click(object sender,
    EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAcciónConMax = "Consulta";
    // Se activan y desactivan los botones correspondientes
    desactivarAccionesConceptos();
}

private void butAltaConMax_Click(object sender, EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAcciónConMax = "Alta";
    // Se activan y desactivan los botones correspondientes
    desactivarAccionesConceptos();
}

private void butBajaConMax_Click(object sender, EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAcciónConMax = "Baja";
```

```
// Se activan y desactivan los botones correspondientes  
desactivarAccionesConceptos();  
}  
  
private void butModificacionConMax_Click(object sender,  
EventArgs e)  
{  
    // Se indica la acción que hay que realizar  
    saccionConMax = "Modificación";  
    // Se activan y desactivan los botones correspondientes  
    desactivarAccionesConceptos();  
}
```

El botón *Cancelar*, como es habitual, vaciará la acción que hay que ejecutar y colocará los botones en la posición inicial, tal y como se muestra a continuación:

```
private void butCancelarConMax_Click(object sender,  
EventArgs e)  
{  
    // Se cancela la acción que hay que realizar  
    saccionConMax = "";  
    // Se activan y desactivan los botones correspondientes  
    activarAccionesConceptos();  
}
```

Por supuesto, el botón *Aceptar* es el que tiene la mayor complejidad de este grupo, puesto que va a ejecutar las correspondientes acciones. Lo mejor es ver primero en detalle qué acciones tiene que ir ejecutando. Dichas acciones son las de la siguiente lista:

- Si se trata de la acción de consulta, se ejecuta la orden de consulta usando el concepto y el año, para así obtener el máximo, que se escribe en la caja de texto correspondiente.
- Si se trata de la acción para dar de alta un elemento, directamente se ejecuta la consulta correspondiente usando todos los elementos de la ventana. Además, se inicializan después dichos elementos.
- Si se trata de la acción para dar de baja, también se ejecuta la orden de eliminación, usando el concepto y el año, puesto que ya identifican el elemento en cuestión. Después se inicializan los elementos de la ventana.

5. Proyectos complementos de programación

- Y en el supuesto de tratarse de la acción de modificación, se modifica la cantidad usando el concepto y el año. También en este caso se inicializan los elementos de la ventana.
- En todas las situaciones se hace lo mismo que al cancelar, primero se vacía la acción que hay que ejecutar y después se habilitan los botones como en su posición inicial.

Todo el código necesario para la ejecución de estas acciones se puede ver en el siguiente evento *Click* del botón *Aceptar*:

```
private void butAceptarConMax_Click(object sender,
    EventArgs e)
{
    // Se chequea la acción que hay que ejecutar
    if (sAccionConMax == "Consulta")
    {
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM tabTopeConceptos "
            + "WHERE tcoConcepto="
            + (lstConceptoMax.SelectedIndex+1) + " "
            + "AND tcoAno=" + cboAnoMax.Text;
        OleDbCommand oleComando = new OleDbCommand(sConsulta,
            conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno válido
        if (oleReader.Read())
        {
            // Se modifican las cajas de texto con los datos
            // del registro leido
            txtTopeMax.Text =
                Convert.ToString(oleReader["tcoTope"]);
        }
        else
        {
            // Si no hay ningún registro, se indica con un
            // mensaje
            MessageBox.Show(
                "El registro introducido no existe");
        }
    }
    else
    {
        if (sAccionConMax == "Alta")
        {
```

```
try
{
    // Se realiza la inserción
    String sConsulta = "INSERT INTO
        tabTopeConceptos (tcoConcepto, tcoAno,
        tcoTope) "
        + "VALUES (" +
        (lstConceptoMax.SelectedIndex + 1) + ",
        "
        + cboAnoMax.Text
        + ", " + txtTopeMax.Text + ")";
    OleDbCommand oleComando = new OleDbCommand();
    oleComando.CommandType = CommandType.Text;
    oleComando.CommandText = sConsulta;
    oleComando.Connection = conBD;
    // Se ejecuta el comando de inserción
    oleComando.ExecuteNonQuery();
    // Se borran las cajas de texto
    lstConceptoMax.SelectedIndex = 0;
    cboAnoMax.SelectedIndex = 0;
    txtTopeMax.Text = "0";
}
catch (Exception ex)
{
    // Si se ha producido un error, se muestra un
    // mensaje
    MessageBox.Show("Error en la inserción");
}
else
{
    if (sAccionConMax == "Baja")
    {
        try
        {
            // Se realiza la eliminación
            String sConsulta =
                "DELETE FROM tabTopeConceptos "
                + "WHERE tcoConcepto="
                + (lstConceptoMax.SelectedIndex +
                1)
                + " "
                + "AND tcoAno=" + cboAnoMax.Text;
            OleDbCommand oleComando =
                new OleDbCommand();
            oleComando.CommandType = CommandType.Text;
            oleComando.CommandText = sConsulta;
            oleComando.Connection = conBD;
            // Se ejecuta el comando de eliminación
        }
    }
}
```

5. Proyectos complementos de programación

```
// Se borran las cajas de texto
lstConceptoMax.SelectedIndex = 0;
cboAnoMax.SelectedIndex = 0;
txtTopeMax.Text = "0";
}
catch (Exception ex)
{
    // Si se ha producido un error, se muestra
    // un mensaje
    MessageBox.Show(
        "Error en la eliminación");
}
else
{
    try
    {
        // Se realiza la modificación
        String sConsulta =
            "UPDATE tabTopeConceptos "
            + "SET tcoTope=" + txtTopeMax.Text
            + " "
            + "WHERE tcoConcepto="
            + (lstConceptoMax.SelectedIndex +
                1)
            + " "
            + "AND tcoAno=" + cboAnoMax.Text;
        OleDbCommand oleComando =
            new OleDbCommand();
        oleComando.CommandType = CommandType.Text;
        oleComando.CommandText = sConsulta;
        oleComando.Connection = conBD;
        // Se ejecuta el comando de modificación
        oleComando.ExecuteNonQuery();
        // Se borran las cajas de texto
        lstConceptoMax.SelectedIndex = 0;
        cboAnoMax.SelectedIndex = 0;
        txtTopeMax.Text = "0";
    }
    catch (Exception ex)
    {
        // Si se ha producido un error, se muestra
        // un mensaje
        MessageBox.Show(
            "Error en la modificación");
    }
}
}
```

```
sAccionConMax = "";
// Se activan y desactivan los botones correspondientes
activarAccionesConceptos();
}
```

En la zona derecha de la ventana se tiene el grupo *Subconceptos*, el cual es similar al anterior, pero con un elemento más: la lista de subconceptos. Para llenar dicha lista se depende del elemento que haya sido seleccionado en la lista de conceptos. Ese relleno se provoca en el evento de cambio de índice en dicha lista de conceptos, tal y como se aprecia aquí:

```
private void lstConceptoSubMax_SelectedIndexChanged(
    object sender, EventArgs e)
{
    // Se actualiza la lista de subconceptos en función del
    // concepto elegido
    leerSubconceptosSubMax();
}
```

En esta zona derecha también se tienen los cuatro botones habituales para la gestión. En cada botón se debe marcar la acción que se quiere ejecutar y también se deben deshabilitar los botones principales para habilitar los de *Aceptar* y *Cancelar*. El código para los cuatro botones principales debe quedar como se muestra a continuación:

```
private void butConsultaSubMax_Click(object sender,
    EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAccionSubMax = "Consulta";
    // Se activan y desactivan los botones correspondientes
    desactivarAccionesSubconceptos();
}

private void butAltaSubMax_Click(object sender, EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAccionSubMax = "Alta";
    // Se activan y desactivan los botones correspondientes
    desactivarAccionesSubconceptos();
}

private void butBajaSubMax_Click(object sender, EventArgs e)
{
    // Se indica la acción que hay que realizar
}
```

5. Proyectos complementos de programación

```
sAccionSubMax = "Baja";
// Se activan y desactivan los botones correspondientes
desactivarAccionesSubconceptos();
}

private void butModificacionSubMax_Click(object sender,
EventArgs e)
{
    // Se indica la acción que hay que realizar
    sAccionSubMax = "Modificación";
    // Se activan y desactivan los botones correspondientes
    desactivarAccionesSubconceptos();
}
```

El botón *Cancelar* de esta parte también vaciará la acción pendiente de ejecutar y colocará los botones en la posición inicial, tal y como se puede ver en este código:

```
private void butCancelarSubMax_Click(object sender,
EventArgs e)
{
    // Se termina la acción realizada
    sAccionSubMax = "";
    // Se activan y desactivan los botones correspondientes
    activarAccionesSubconceptos();
}
```

Y para terminar este apartado, queda el botón *Aceptar*, que es el que va a ejecutar las acciones correspondientes. Como en el apartado anterior, primero se van a describir en detalle las acciones que se deben ir ejecutando. Dichas acciones son las de esta lista:

- Si se trata de la acción de consulta, se ejecuta la orden de consulta usando el concepto, el subconcepto y el año, para así obtener el máximo, que se escribe en la caja de texto correspondiente.
- Si se trata de la acción para dar de alta un elemento, directamente se ejecuta la consulta correspondiente usando todos los elementos de la ventana. Además, se inicializan después dichos elementos.
- Si se trata de la acción para dar de baja, también se ejecuta la orden de eliminación, usando el concepto, el subconcepto y el año, puesto que ya identifican el elemento en cuestión. Después se inicializan los elementos de la ventana.

- Y en el supuesto de tratarse de la acción de modificación, se modifica la cantidad usando el concepto, el subconcepto y el año. También en este caso se inicializan los elementos de la ventana.
- En todas las situaciones se hace lo mismo que al cancelar, primero se vacía la acción pendiente de ejecutar y después se habilitan los botones como en su posición inicial.

Todo el código necesario para la ejecución de estas acciones se puede ver en el evento *Click* del botón *Aceptar*:

```
private void butAceptarSubMax_Click(object sender,
EventArgs e)
{
    // Se chequea la acción que hay que ejecutar
    if (sAcciónSubMax == "Consulta")
    {
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM tabTopeSubconceptos "
            + "WHERE tscConcepto="
            + (lstConceptoSubMax.SelectedIndex + 1) + " "
            + "AND tscSubconcepto="
            + (lstSubconceptoSubMax.SelectedIndex + 1) +
            " "
            + "AND tscAño=" + cboAñoSubMax.Text;
        OleDbCommand oleComando = new OleDbCommand(sConsulta,
            conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno válido
        if (oleReader.Read())
        {
            // Se modifican las cajas de texto con los datos
            // del registro leído
            txtTopeSubMax.Text =
                Convert.ToString(oleReader["tscValorTope"]);
        }
        else
        {
            // Si no hay ningún registro, se indica con un
            // mensaje
            MessageBox.Show(
                "El registro introducido no existe");
        }
    }
    else
    {
        // Se limpian los campos
        txtTopeSubMax.Text = "";
        txtSubconceptoSubMax.Text = "";
        txtAñoSubMax.Text = "";
    }
}
```

5. Proyectos complementos de programación

```
{  
    if (sAccionSubMax == "Alta")  
    {  
        try  
        {  
            // Se realiza la inserción  
            String sConsulta =  
                "INSERT INTO tabTopeSubconceptos "  
                + "(tscConcepto, tscSubconcepto, tscA-  
                no,  
                tscValorTope) "  
                + "VALUES ("  
                + (lstConceptoSubMax.SelectedIndex + 1)  
                + ", "  
                + (lstSubconceptoSubMax.SelectedIndex +  
                1)  
                + ", " + cboAnoSubMax.Text  
                + ", " + txtTopeSubMax.Text + ")";  
            OleDbCommand oleComando = new OleDbCommand();  
            oleComando.CommandType = CommandType.Text;  
            oleComando.CommandText = sConsulta;  
            oleComando.Connection = conBD;  
            // Se ejecuta el comando de inserción  
            oleComando.ExecuteNonQuery();  
            // Se borran las cajas de texto  
            lstConceptoSubMax.SelectedIndex = 0;  
            lstSubconceptoSubMax.SelectedIndex = 0;  
            cboAnoSubMax.SelectedIndex = 0;  
            txtTopeSubMax.Text = "0";  
        }  
        catch (Exception ex)  
        {  
            // Si se ha producido un error, se muestra un  
            // mensaje  
            MessageBox.Show("Error en la inserción");  
        }  
    }  
    else  
    {  
        if (sAccionSubMax == "Baja")  
        {  
            try  
            {  
                // Se realiza la eliminación  
                String sConsulta =  
                    "DELETE FROM tabTopeSubconceptos "  
                    + "WHERE tscConcepto="  
                    + (lstConceptoSubMax.SelectedIndex + 1);  
            }  
        }  
    }  
}
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
+ 1) + " "
+ "AND tscSubconcepto="
+ (lstSubconceptoSubMax.SelectedIndex
+ 1) + " "
+ "AND tscAno=" + cboAnoSubMax.Text;
OleDbCommand oleComando =
    new OleDbCommand();
oleComando.CommandType = CommandType.Text;
oleComando.CommandText = sConsulta;
oleComando.Connection = conBD;
// Se ejecuta el comando de eliminación
oleComando.ExecuteNonQuery();
// Se borran las cajas de texto
lstConceptoSubMax.SelectedIndex = 0;
lstSubconceptoSubMax.SelectedIndex = 0;
cboAnoSubMax.SelectedIndex = 0;
txtTopeSubMax.Text = "0";
}
catch (Exception ex)
{
    // Si se ha producido un error, se muestra
    // un mensaje
    MessageBox.Show(
        "Error en la eliminación");
}
}
else
{
try
{
    // Se realiza la modificación
    String sConsulta =
        "UPDATE tabTopeSubconceptos "
        + "SET tscValorTope="
        + txtTopeSubMax.Text + " "
        + "WHERE tscConcepto="
        + (lstConceptoSubMax.SelectedIndex
        + 1) + " "
        + "AND tscSubconcepto="
        + (lstSubconceptoSubMax.SelectedIndex
        + 1) + " "
        + "AND tscAno=" + cboAnoSubMax.Text;
    OleDbCommand oleComando =
        new OleDbCommand();
    oleComando.CommandType = CommandType.Text;
    oleComando.CommandText = sConsulta;
    oleComando.Connection = conBD;
    // Se ejecuta el comando de modificación
    oleComando.ExecuteNonQuery();
```

5. Proyectos complementos de programación

```
    1stConceptoSubMax.SelectedIndex = 0;
    1stSubconceptoSubMax.SelectedIndex = 0;
    cboAnoSubMax.SelectedIndex = 0;
    txtTopeSubMax.Text = "0";
}
catch (Exception ex)
{
    // Si se ha producido un error, se muestra
    // un mensaje
    MessageBox.Show(
        "Error en la modificación");
}
}
}
}
// Se termina la acción realizada
sAccionSubMax = "";
// Se activan y desactivan los botones correspondientes
activarAccionesSubconceptos();
}
```

Por último, quedan una serie de funciones de la aplicación en general que también deben ser comentadas. La primera es la que se hace cuando se redimensiona la ventana. Aunque hay elementos que tienen un tamaño fijo, es conveniente modificar el tamaño de otros, de manera que al redimensionar la ventana, se queden con unas proporciones correctas. Para ello se utiliza el evento *Resize* de la propia ventana. Los elementos cuyo tamaño va a ser modificado son los tabuladores generales, el gráfico de la ventana principal y los grupos del segundo tabulador. Este código se puede ver a continuación:

```
private void frmContabilidad_Resize(object sender,
    EventArgs e)
{
    // Se pone la anchura de los tabuladores generales
    tabControles.Width = this.Width - 15;
    // Se modifica el tamaño de la gráfica según convenga
    graDatos.Width = this.Width - 280;
    grdDatos.Width = this.Width - 40;
    // Se modifica el tamaño de los grupos
    grpConceptos.Width = (this.Width - 60) / 2;
    grpConceptos.Height = this.Height - 140;
    grpSubconceptos.Width = grpConceptos.Width;
    grpSubconceptos.Height = grpConceptos.Height;
    grpSubconceptos.Left = grpConceptos.Left
        + grpConceptos.Width + 20;
}
```

Para terminar, queda comentar lo que se hace en el cierre de la aplicación. Por una parte, se debe cerrar la conexión con la base de datos, para lo cual se utiliza una función creada para tal fin que primero verifica si existe una conexión abierta y, dado el caso, la finaliza. Dicha función es:

```
public void cerrarConexion()
{
    // Se chequea si está la conexión abierta
    if (conBD.State == ConnectionState.Open)
    {
        // Se cierra la conexión
        conBD.Close();
    }
}
```

En el cierre de la aplicación, lo único que se debe hacer es llamar a la función anterior y después cerrar el propio formulario. Ello está escrito al pulsar el botón de salida, tal y como se puede ver aquí:

```
private void butSalir_Click(object sender, EventArgs e)
{
    // Se cierra la conexión antes de terminar
    cerrarConexion();
    // Finaliza el programa
    this.Close();
}
```

Proyecto completo con bibliotecas

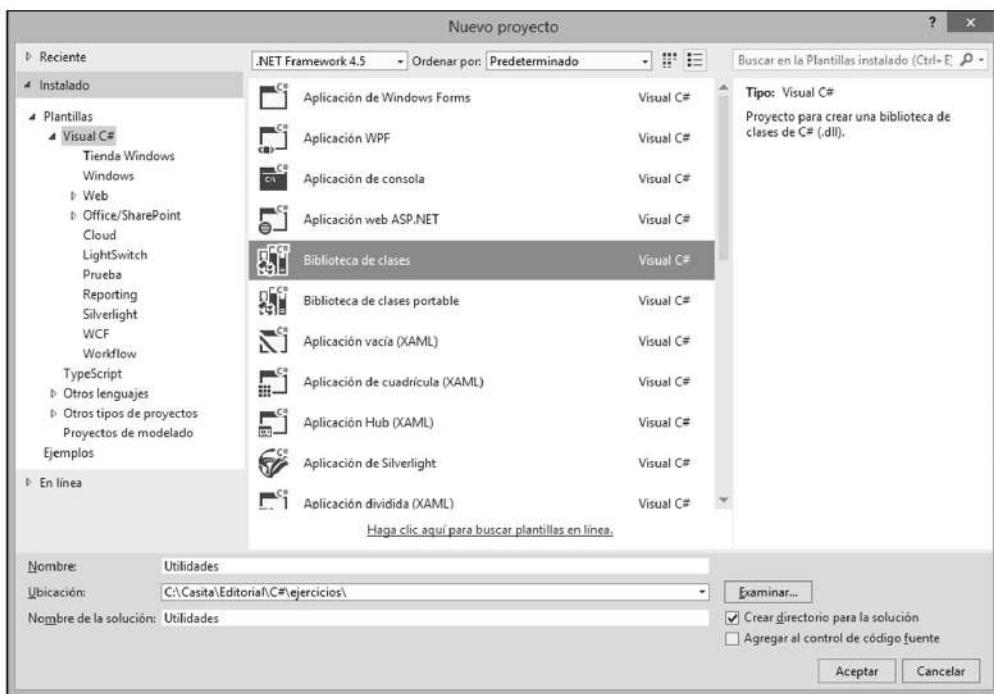
Otro tipo muy común a la hora de realizar proyectos es el de las bibliotecas. Cuando se van a crear funciones u objetos que pueden ser utilizados en diferentes proyectos, es conveniente que se creen bibliotecas que después puedan ser accesibles desde cualquiera de los nuevos proyectos de una manera organizada. De esta forma, cuando se modifique el contenido

5. Proyectos complementos de programación

de una función dentro de una biblioteca, automáticamente ese cambio se realizará en todos los proyectos que la utilicen.

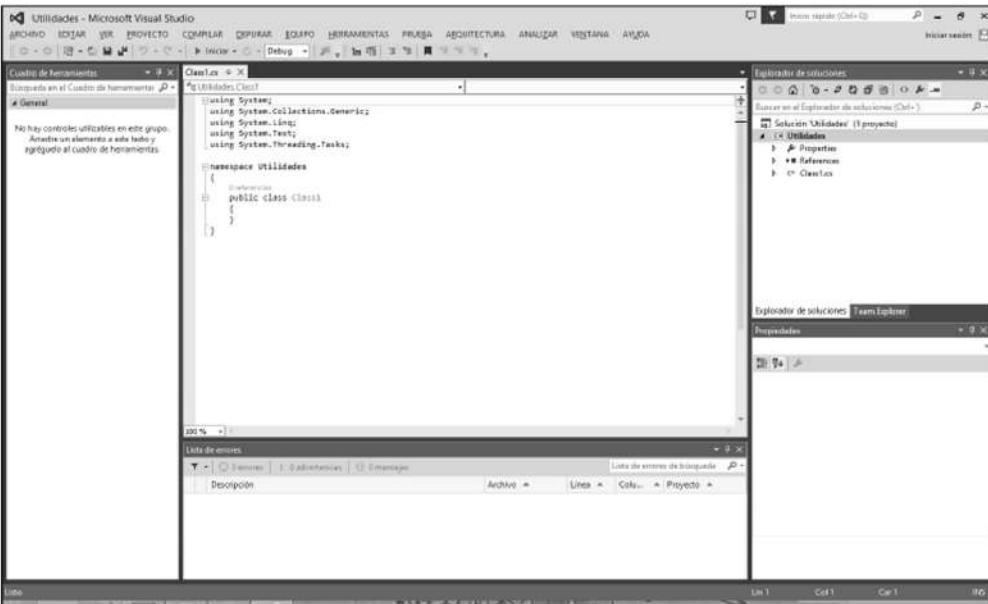
En el sistema operativo *Windows*, las bibliotecas son archivos con la extensión *DLL* y las siglas vienen de *Dynamic Link Library* o *Biblioteca de Vínculos Dinámicos*. En su definición genérica se indica que son archivos ejecutables que permiten a los programas compartir código y otros recursos necesarios para realizar ciertas tareas. El propio sistema operativo proporciona muchas bibliotecas de este tipo, que pueden ser utilizadas en las aplicaciones.

Cuando se crea un nuevo proyecto, el tipo que hay que elegir es *Biblioteca de clases*. En este caso se va a crear una biblioteca llamada *Utilidades*. La selección del proyecto se debe hacer tal y como se muestra en la siguiente imagen.



Desarrollo de aplicaciones C# con Visual Studio .NET

Cuando se abre el proyecto aparece la clase vacía para empezar a rellenarla. Se puede ver en esta imagen:



El nombre de la clase por defecto es *Class1*, aunque se puede modificar y asignarle un nombre cualquiera, que será el que después se usará desde los proyectos. Es recomendable realizar dicha modificación.

Para esta práctica se van a crear una serie de funciones que se pueden agrupar en dos grupos diferenciados, uno relacionado con las bases de datos y otro relacionado con las licencias que se puedan controlar en una aplicación. En la siguiente lista de tablas se describen las funciones que se quieren realizar, qué parámetros deben tener, qué datos van a obtener de salida y qué procesos deben ejecutar dentro de ellas. El contenido de cada tabla va a ser como se describe a continuación.

5. Proyectos complementos de programación

Función	Descripción
Parámetros	Salida

La lista de las funciones relacionadas con la base de datos es la que se especifica en estas tablas:

<i>abrirConexion</i>	Abre una conexión con la base de datos Access que se pasa como parámetro.
<i>sBaseDatos – String</i>	Conexión abierta si se ha podido abrir o valor nulo si ha habido algún problema.

<i>cerrarConexion</i>	Cierra una conexión con la base de datos Access que se pasa como parámetro.
<i>conBD – OleDbConnecion</i>	Nada.

<i>leerTabla</i>	Lee el contenido de una tabla, extrae todos los campos que contiene y devuelve una lista con todos los contenidos en un <i>array</i> de elementos en el que cada elemento es la concatenación de todos los campos.
<i>sTabla – String</i> <i>sCampos - String</i> <i>conBD – OleDbConnecio</i>	<i>Array</i> de elementos con los contenidos de la tabla. Si no hubiera datos en la tabla, se devolvería el <i>array</i> vacío.

Y la lista de las funciones relacionadas con la parte de licenciamiento es la que se especifica en estas otras tablas:

<i>MD5</i>	Convierte el dato introducido en una cadena codificada con el algoritmo <i>MD5</i> .
<i>sOriginal – String</i>	Cadena de caracteres con el dato ya codificado.

<i>crearFichero</i>	Crea un archivo de propiedades en el que se guardan los siguientes datos: <ul style="list-style-type: none"> • Nombre de la aplicación. • Versión de la aplicación. • Clave de la CPU codificada en <i>MD5</i>. • Clave del disco duro codificada en <i>MD5</i>.
<i>appName – String</i> <i>appVersion – String</i> <i>outputFile – String</i>	Archivo de texto de salida con el nombre indicado en el parámetro y el contenido especificado.

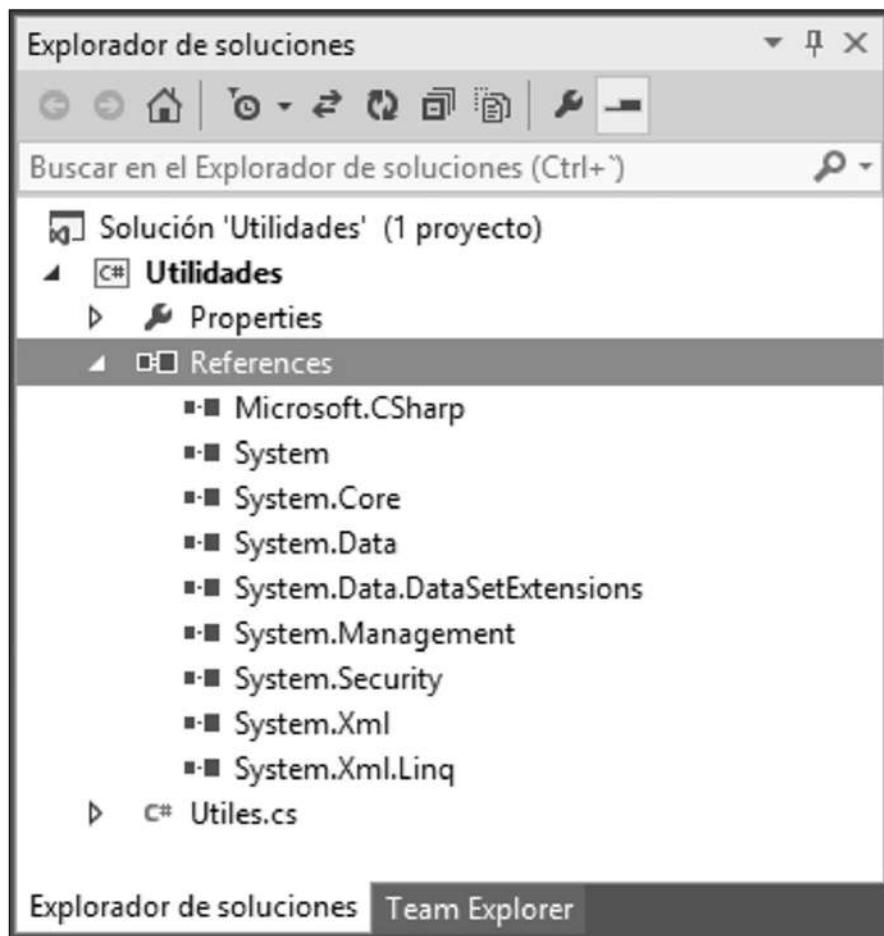
<i>chequearElemento</i>	Chequea si el elemento que se le indica coincide con el contenido ya codificado anteriormente.
<i>sElemento – String</i> <i>sElementoGuardado – String</i>	<i>True</i> si coinciden y <i>False</i> si no coinciden.

Antes de ver el código, hay que indicar que son necesarias una serie de directivas *using* al comienzo de la clase. Dichas directivas son las de la siguiente lista:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data;
using System.Data.OleDb;
using System.IO;
using System.Security.Cryptography;
using System.Security.Cryptography.Xml;
using System.Xml;
using System.Management;
```

Además, para poder utilizarlas, es necesario agregar al proyecto un par de referencias. Para ello hay que ir al explorador de soluciones y agregarlas en la pestaña de *.NET*. Deben quedar como en la siguiente imagen.

5. Proyectos complementos de programación



Una vez que se tengan todas las referencias colocadas, ya se puede generar el código para cada una de las funciones especificadas. De nuevo se va a ver dicho código distinguiendo entre las funciones para las bases de datos y las funciones para el licenciamiento. El de la parte de las bases de datos es el que se puede ver en las siguientes funciones:

```
public OleDbConnection abrirConexion(String sBD)
{
    // Se genera la variable para la conexión
    OleDbConnection conBD = new OleDbConnection();
    // Se crea el string de conexión a la base de datos
    conBD.ConnectionString =
        @"Provider=Microsoft.ACE.OLEDB.12.0; Data Source=" +
    sBD
    . . .
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
try
{
    // Se abre la base de datos
    conBD.Open();
}
catch (Exception ex)
{
    // Se devuelve el valor nulo
    conBD = null;
}
return conBD;
}

public void cerrarConexion(OleDbConnection conBD)
{
    // Se chequea si está la conexión abierta
    if (conBD.State == ConnectionState.Open)
    {
        // Se cierra la conexión
        conBD.Close();
    }
}

private String[] leerTabla(String sTabla, String sCampos,
    OleDbConnection conBD)
{
    // Se crea la matriz para los elementos
    String[] sElementos = null;
    try
    {
        // Antes de empezar hay que saber el número de
        // elementos que va a tener la matriz
        int iCantidad = 0;
        // Se realiza la consulta
        String sConsulta = "SELECT COUNT(*) AS Cantidad "
            + "FROM " + sTabla;
        OleDbCommand oleComando = new OleDbCommand(sConsulta,
conBD);
        OleDbDataReader oleReader = oleComando.ExecuteReader();
        // Se lee el resultado para ver la cantidad de
        // elementos
        while (oleReader.Read())
        {
            iCantidad = Convert.ToInt32(oleReader["Canti-
dad"]);
        }
        // Se crea la matriz vacía con el número de elementos
        // deseado
    }
}
```

5. Proyectos complementos de programación

```
// Se realiza la consulta completa
sConsulta = "SELECT " + sCampos + " "
+ "FROM " + sTabla;
oleComando = new OleDbCommand(sConsulta, conBD);
oleReader = oleComando.ExecuteReader();
// Se obtiene la lista de campos
String[] sListaCampos = sCampos.Split(',');
// Se leen los resultados
int iElemento = 0;
while (oleReader.Read())
{
    String sElemento = "";
    // Se añade cada campo al elemento de la línea
    for (int iCont = 0; iCont < sListaCampos.Length;
iCont++)
    {
        sElemento = sElemento + Convert.To-
String(oleReader[
            sListaCampos[iCont].Trim()]) + " ";
    }
    // Se añade el elemento a la lista
    sElementos[iElemento] = sElemento;
    iElemento++;
}
}
catch (Exception ex)
{
    // Si salta una excepción,no se hace nada específico y
    // se termina la ejecución
}
// Se devuelve la lista con los resultados
return sElementos;
}
```

Y el código de las funciones que se necesitan para realizar la parte del licenciamiento es el siguiente:

```
public string MD5(string sOriginal)
{
    // Declaración de variables
    Byte[] originalBytes;
    Byte[] encodedBytes;
    MD5 md5;
    // Se instancia MD5CryptoServiceProvider para obtener los
    // bytes del dato original
    md5 = new MD5CryptoServiceProvider();
    originalBytes = ASCIIEncoding.Default.GetBytes(sOriginal);
}
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
// Se codifican usando MD5
encodedBytes = md5.ComputeHash(originalBytes);
// Se convierten los bytes codificados a una cadena
// legible
return BitConverter.ToString(encodedBytes);
}

public void crearFichero(String appName, String appVersion,
    String outputFile)
{
    // Se chequean el número de serie del disco duro y de la
    // CPU
    string cpuInfo = String.Empty;
    string temp = String.Empty;
    // Se obtiene el dato del procesador
    ManagementClass mc = new ManagementClass("Win32_Proces-
sor");
    ManagementObjectCollection moc = mc.GetInstances();
    foreach (ManagementObject mo in moc)
    {
        if ((cpuInfo == String.Empty))
        {
            cpuInfo = mo.Properties["ProcessorId"].Value.To-
String();
        }
    }
    // Se obtiene el dato del disco duro
    DirectoryInfo currentDir =
        new DirectoryInfo(Environment.CurrentDirectory);
    string path = string.Format("win32_logicaldisk.devic-
eid=\\"{0}\\\"", 
        currentDir.Root.Name.Replace("\\\\", ""));
    ManagementObject disk = new ManagementObject(path);
    disk.Get();
    string hddInfo = disk["VolumeSerialNumber"].ToString();
    // Se escribe el contenido del fichero
    StreamWriter file = new StreamWriter(outputFile, false);
    file.WriteLine("app.name=" + appName);
    file.WriteLine("app.version=" + appVersion);
    file.WriteLine("cpu.key=" + MD5(cpuInfo));
    file.WriteLine("hdd.key=" + MD5(hddInfo));
    // Se cierra el fichero
    file.Close();
}

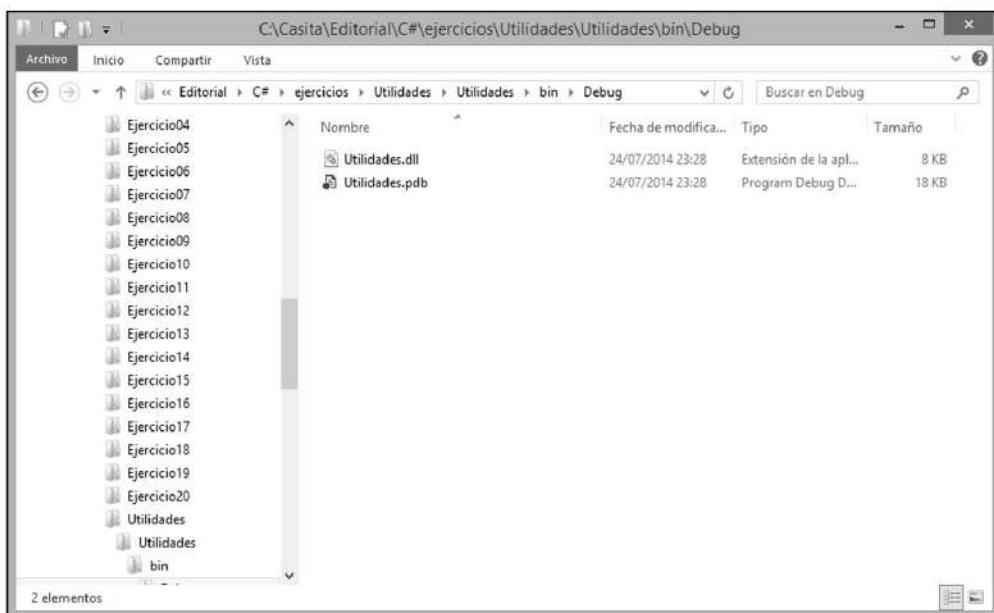
private Boolean chequearElemento(String sElemento,
    String sElementoGuardado)
{
    // Se chequea si el elemento buscado es como el que se
```

5. Proyectos complementos de programación

```
if (MD5(sElemento) == sElementoGuardado)
{
    // Sí que coinciden, así que se devuelve True
    return true;
}
else
{
    // No coinciden, así que se devuelve False
    return false;
}
```

Una vez que esté ya todo el código escrito, es el momento de generar la biblioteca. Hay que recordar que este tipo de proyecto no es para ejecutar, así que lo único que hay que hacer es compilar el contenido para generar el archivo *DLL* que después podrá ser usado en otros proyectos. Para ello hay que ir al menú *COMPILAR* y pulsar en la opción *Compilar solución*. Si no hay ningún problema, en la barra de estado de la parte inferior aparecerá el mensaje *Compilación correcta*.

Una vez terminado el archivo, hay que ir a la carpeta *bin* dentro del proyecto y ahí, en la subcarpeta *Debug* o en la subcarpeta *Release*, en función de lo que se tenga definido en el proyecto, habrá aparecido el archivo con la nueva biblioteca, tal y como se aprecia en la siguiente imagen.



Una vez generada la biblioteca, puede usarse en cualquier otro proyecto, tras añadirla al mismo. Todas las funciones contenidas en la biblioteca se podrán usar como si estuvieran definidas en el propio proyecto.

Instalación de aplicaciones

CAPÍTULO
6

Las aplicaciones *ClickOnce*

El principio básico de *ClickOnce* es permitir desplegar aplicaciones en el sistema operativo *Windows*. Su objetivo es resolver los problemas que solían tener los instaladores anteriores:

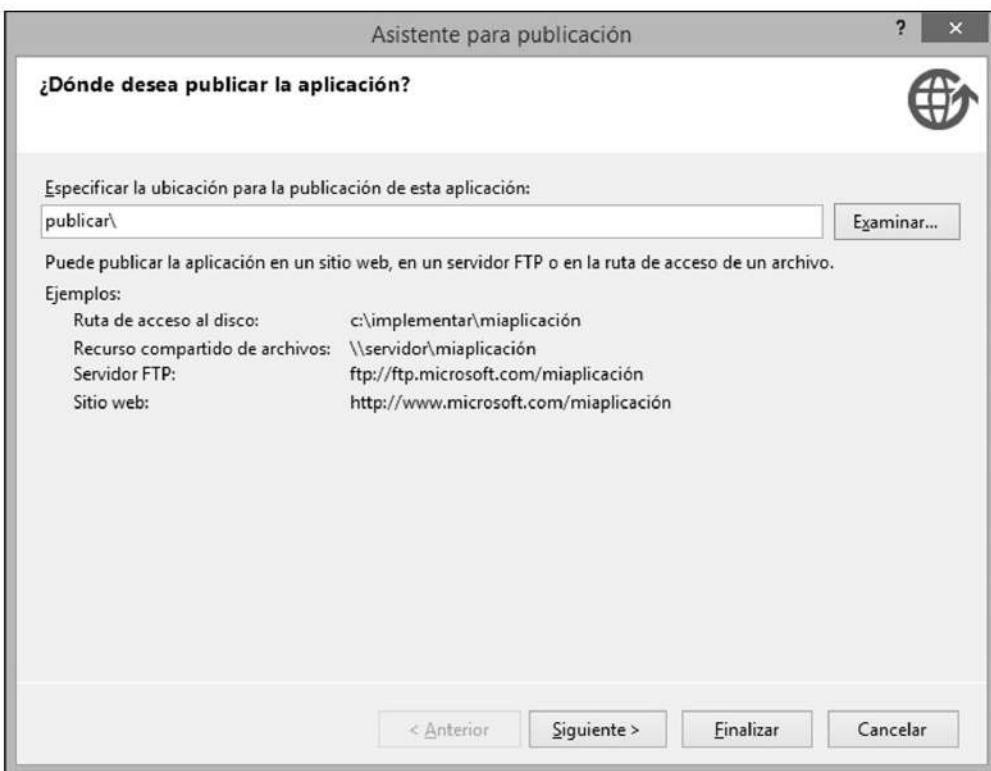
- La dificultad a la hora de actualizar una aplicación anteriormente creada.
- El impacto de la nueva aplicación en el ordenador del usuario.
- Los permisos de administrador para instalar las aplicaciones.

Ahora este nuevo sistema de instalación permite resolver dichos problemas. De entrada, la aplicación se instala por usuario, no por equipo, con lo que no necesita los privilegios de administrador para instalarse. Por otra parte, cada aplicación instalada de este tipo está aislada de las demás, así que nunca puede interferir en ellas. Por último, estas aplicaciones usan el CAS o seguridad de acceso a código, lo que garantiza que no se van a llamar a funciones del sistema desde la web, con lo que se obtiene un sistema seguro.

Publicación de proyectos

En esta versión se ha cambiado completamente la forma de hacer las publicaciones de los proyectos con respecto a la versión del 2010. Lo mejor es explicarlo paso a paso, para que quede claro cómo se debe realizar un instalador desde el principio, partiendo de un proyecto ya existente.

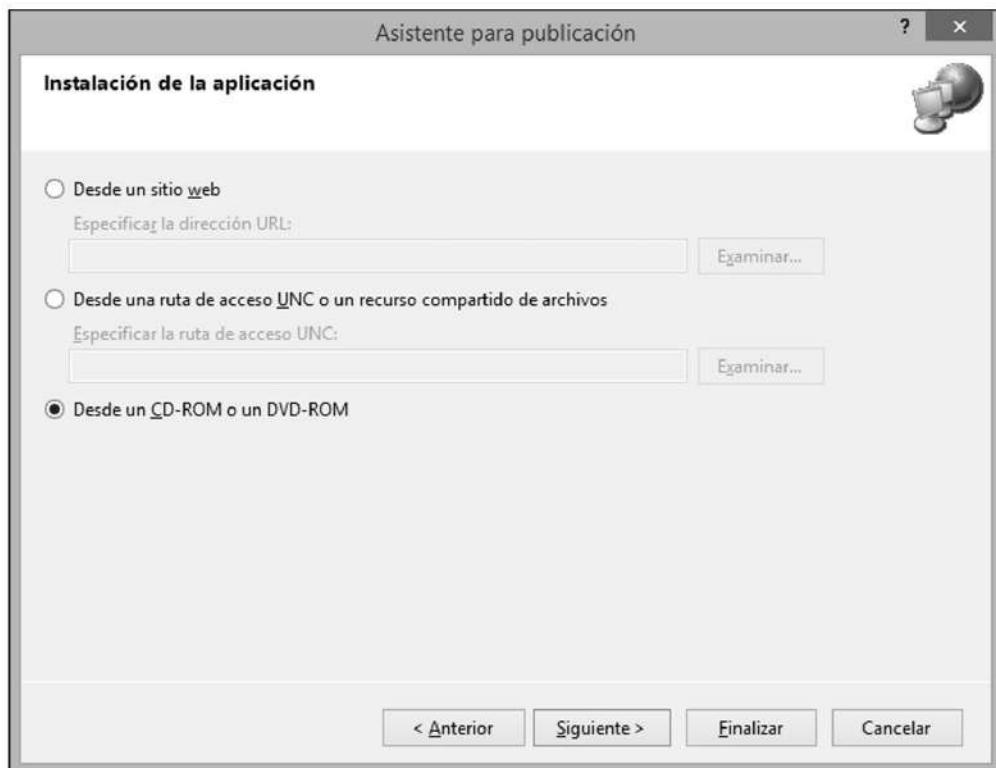
En este caso se puede partir del proyecto *Contabilidad* que se ha visto en el proyecto completo del apartado 5 de este libro. Lo que hay que hacer es abrir dicho proyecto, ir a la pestaña del *Explorador de soluciones* y pulsar con el botón derecho del ratón sobre el nombre del propio proyecto. La opción que se debe elegir es *Publicar*. Al hacerlo se abre la siguiente ventana:



6. Instalación de aplicaciones

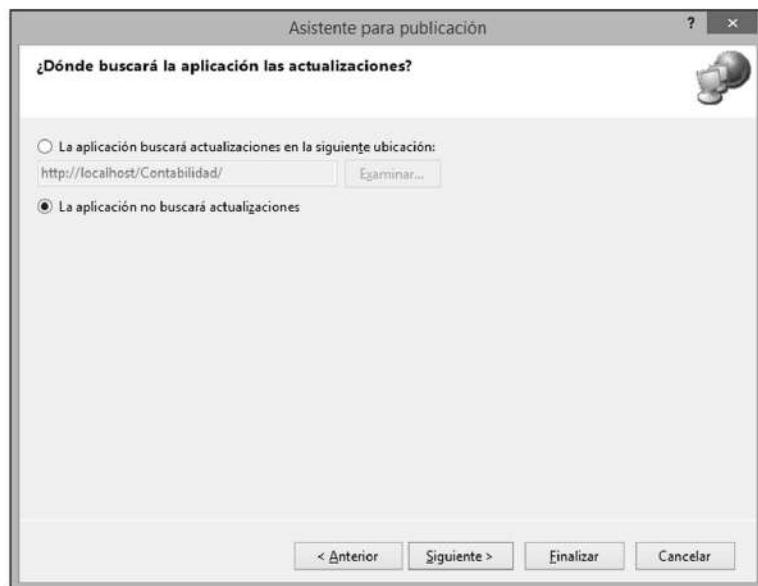
Como se puede apreciar en la imagen, se tiene la posibilidad de dejar el instalador en un sitio web, en un servidor de *FTP* o en una ruta de un disco duro o de cualquier unidad externa. Por defecto aparece una subcarpeta dentro del proyecto de la aplicación.

Cuando se pulsa el botón *Siguiente*, se llega a la siguiente ventana:

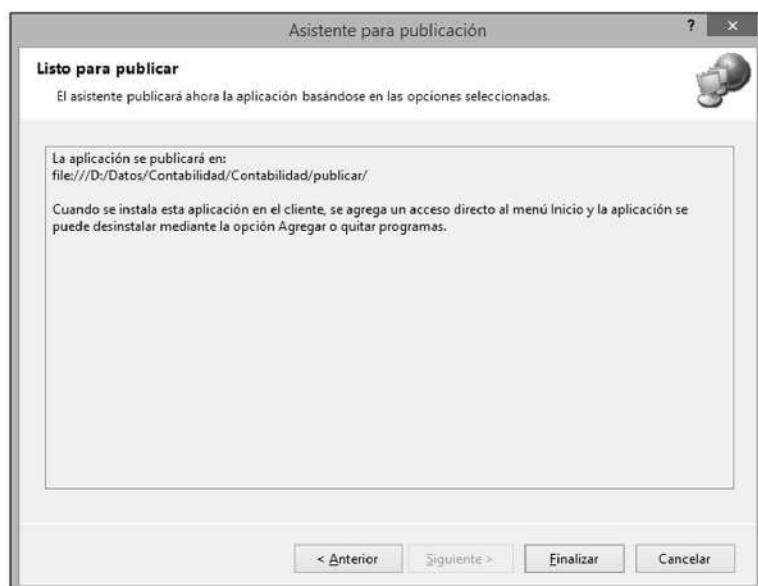


A la hora de alojar la aplicación, se dan tres opciones. La primera es instalarla desde un sitio web, para lo cual se va a pedir la ruta de dicho sitio. La segunda es una ruta de acceso compartido, para lo cual se solicita una ruta en formato de red. Y la tercera y última es que se haga desde un CD o DVD, para lo cual no es necesario indicar ninguna ubicación. En este caso se va a seleccionar la tercera opción y de nuevo se va a pulsar el botón *Siguiente* para continuar. La nueva ventana que aparece es:

Desarrollo de aplicaciones C# con Visual Studio .NET



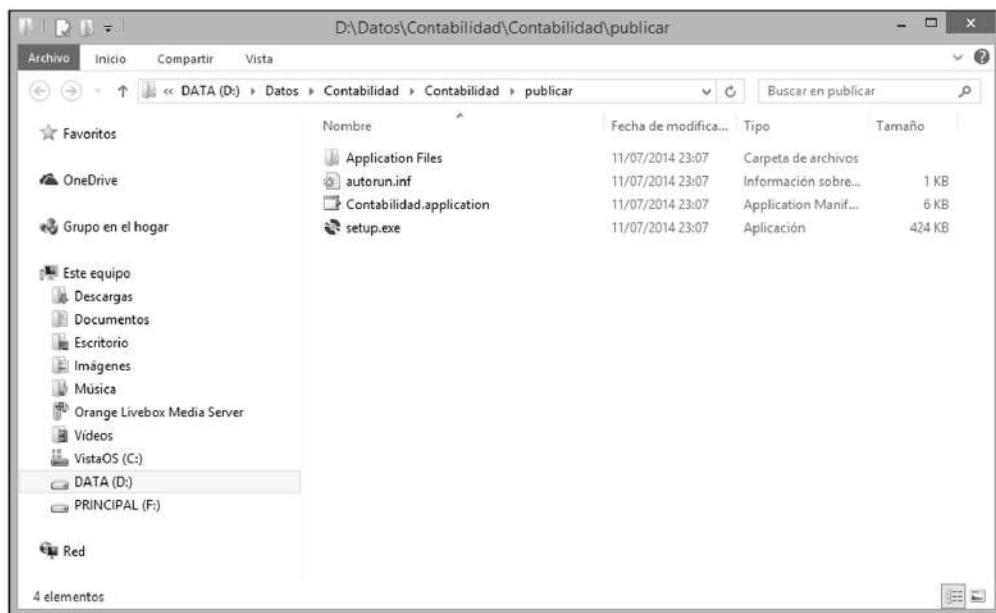
En esta nueva ventana se puede especificar el lugar desde el que la aplicación puede buscar nuevas actualizaciones. Como en este caso es una aplicación ya cerrada y que no se va a modificar, se elige la segunda opción, que le indica que no es necesario que busque nada. De nuevo se pulsa el botón *Siguiente* para seguir con el proceso. Se abre entonces la nueva ventana:



6. Instalación de aplicaciones

Aquí se visualiza que todo esté correcto y si es así, ya se puede pulsar el botón *Finalizar* para terminar el proceso.

A partir de ese momento se comienzan a preparar los archivos del instalador. El proceso durará más o menos tiempo en función de la envergadura del proyecto. Una vez finalizado el proceso, se abrirá automáticamente en el *Explorador de archivos de Windows* la carpeta en la que esté el resultado de lo que se ha generado.



Instalación de proyectos

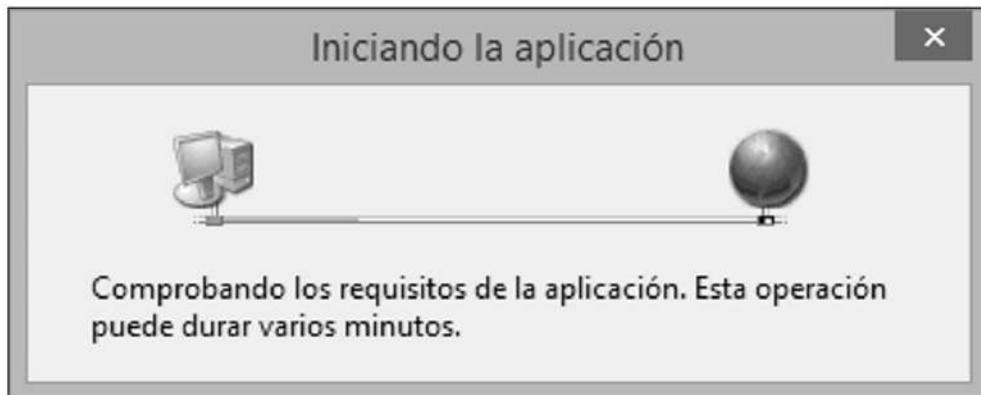
Una vez generado el instalador por completo, se puede probar a realizar la instalación. Para ello, hay que ir a la carpeta en la que se han dejado los archivos de la aplicación y lanzar el archivo *setup.exe*.

Al ejecutar el instalador, primero se muestra una ventana en la que aparece el nombre de la aplicación, el lugar desde el que se está instalando y el nombre del fabricante si lo hubiere. El aspecto que tiene esta primera ventana es el siguiente:

Desarrollo de aplicaciones C# con Visual Studio .NET



El usuario debe pulsar el botón *Instalar* para que comience el proceso de instalación. Lo primero que hace el instalador es chequear los requisitos de instalación. Este proceso durará más o menos tiempo en función del tipo de aplicación. La ventana que aparece en este proceso es:



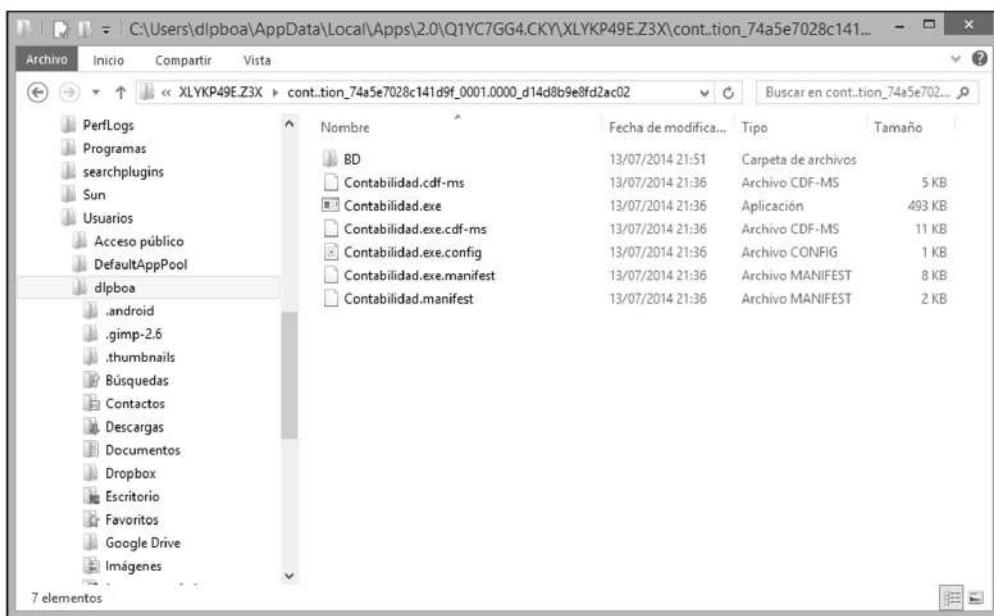
6. Instalación de aplicaciones

Una vez terminado el proceso de instalación, la aplicación se encuentra colocada en una carpeta dentro de los datos de aplicaciones del usuario. Por ejemplo, en este caso se ha instalado en la ruta:

C:\Users\dlpboa\AppData\Local\Apps\2.0\Q1YC7GG4.CKY\XLYKP49E.Z3X\cont..tion_74a5e7028c141d9f_0001.0000_d14d8b9e8fd2ac02

Hay que tener en cuenta que si la aplicación necesita algún elemento externo que no constaba en el instalador, habría que copiarlo manualmente. En este caso se debe crear la carpeta en la que se encuentra la base de datos.

En el *Explorador de archivos* se pueden ver los archivos que se han generado después de realizar la instalación. En este caso debe quedar como se muestra en la siguiente ventana:



Para verificar que la aplicación está correctamente instalada, se puede ir a la lista de aplicaciones y comprobar que se ha creado una nueva aplicación. Como se puede apreciar en la siguiente imagen, aparece un nuevo elemento llamado *Contabilidad*, que además aparece con la marca *NUEVO*, que indica que está recientemente instalado.

Desarrollo de aplicaciones C# con Visual Studio .NET



Soluciones a los ejercicios

CAPÍTULO
7

Ejercicio 2.1

```
class Coche
{
    // Definición de la clase
    public Coche(string marca, string modelo, int vmaxima)
    {
        this.Marca=marca;
        this.Modelo=modelo;
        this.Vmaxima=vmaxima;
        this.Vactual=0;
    }

    public string Marca;
    public string Modelo;
    public string Vmaxima;
    public string Vactual;

    public void Acelera()
    {
        // Se incrementa la velocidad
        Vactual += 5;
        // Se muestra la nueva velocidad
    }
}
```

```
        }
        public void Decelera()
        {
            // Se reduce la velocidad
            Vactual -= 5;
            // Se muestra la nueva velocidad
            Console.WriteLine("Bajando a: " + Vactual + " Km/h");
        }
    }
```

Ejercicio 2.2

```
class CocheConFreno : Coche
{
    public CocheConFreno (string marca, string modelo, int vmaxima) :
        base(marca, modelo, vmaxima) {}

    public void Frena()
    {
        // Se inicializa la velocidad
        Vactual = 0;
        // Se muestra la nueva velocidad
        Console.WriteLine("Frenado a: " + Vactual + " Km/h");
    }
}
```

Ejercicio 2.3

```
class Ordenador
{
    // Definición de la clase
    public Ordenador(string marca, string procesador, int memoria,
                     int disco)
    {
        this.Marca=marca;
        this.Procesador=procesador;
        this.Memoria=memoria;
        this.Disco=disco;
    }
    public string Marca;
```

7. Soluciones a los ejercicios

```
public string Procesador;
public string Memoria;
public string Disco;

public void MayorCapacidad()
{
    // Se incrementa la capacidad del disco duro
    Disco += 100;
    // Se muestra la nueva capacidad
    Console.WriteLine("Nueva capacidad: " + Disco + " Gb");
}

public void MenorCapacidad()
{
    // Se reduce la capacidad del disco duro
    Disco -= 100;
    // Se muestra la nueva capacidad
    Console.WriteLine("Nueva capacidad: " + Disco + " Gb");
}
```

Ejercicio 3.1

```
// Declaración de variables
int[,] matriz1 = new int[5, 5];
int[,] matriz2 = new int[5, 5];
int[,] matrizSuma = new int[5, 5];

// Inicialización de datos
matriz1 = {{5, 6, 4, 2, 8}, {1, 9, 7, 3, 5}, {2, 9, 0, 5, 7}, {1, 6, 8, 4, 9}, {3, 6, 7, 4, 5}};

matriz2 = {{3, 0, 5, 9, 4}, {8, 1, 2, 6, 7}, {5, 9, 4, 8, 3}, {1, 2, 0, 6, 5}, {8, 9, 4, 2, 7}};

// Se realiza la suma de los elementos de ambas matrices
for (byte iCont1=0; iCont1<5 ; iCont1++)
{
    for (byte iCont2=0; iCont2<5 ; iCont2++)
        matrizSuma[iCont1, iCont2] = matriz1[iCont1, iCont2] +
            matriz2[iCont1, iCont2];
}
```

Ejercicio 3.2

```
// Declaración de variables
int[,] matriz1 = new int[5, 5];
int[,] matriz2 = new int[5, 5];
int[,] matrizResta = new int[5, 5];
byte iCont1 = 0;
byte iCont2 = 0;

// Inicialización de datos
matriz1 = {{5, 6, 4, 2, 8}, {1, 9, 7, 3, 5}, {2, 9, 0, 5, 7}, {1,
6, 8, 4, 9}, {3, 6, 7, 4, 5}};

matriz2 = {{3, 0, 5, 9, 4}, {8, 1, 2, 6, 7}, {5, 9, 4, 8, 3}, {1,
2, 0, 6, 5}, {8, 9, 4, 2, 7}};

// Se realiza la resta de los elementos de ambas matrices
while (iCont1<5)
{
    // Cada vuelta debe inicializar la variable del 2º bucle
    iCont2 = 0;
    while (iCont2<5)
    {
        matrizResta[iCont1, iCont2] = matriz1[iCont1, iCont2]-
            matriz2[iCont1, iCont2];
        iCont2++;
    }
    iCont1++;
}
```

Ejercicio 4.1

Lo primero que hay que hacer es el diseño de la ventana. Basta con colocar los mismos objetos que aparecen en el diseño propuesto. Para ello se van a utilizar las siguientes herramientas:

- *Label*: etiquetas que contienen texto fijo.
- *Button*: botones de comando.

7. Soluciones a los ejercicios

Pulsando directamente sobre el fondo del formulario, se abre el evento *Load*, que es el evento principal para el formulario. En dicho evento se deben inicializar los valores de las etiquetas de operandos y operación de esta manera:

```
private void frmEjercicio13_Load(object sender, EventArgs e)
{
    // Se inicializan todas las etiquetas
    lblOperando1.Text = "0";
    lblOperando2.Text = "0";
    lblOperacion.Text = "";
}
```

Como en los ejemplos descritos en el libro, el botón *Salir* debe cerrar la aplicación y para ello, en su evento principal *Click*, debe llamar al método *Close* haciendo referencia al formulario de la siguiente forma:

```
private void butSalir_Click(object sender, EventArgs e)
{
    // Se cierra la aplicación
    this.Close();
}
```

Para los botones numéricos, el código es prácticamente igual, pero cambiando el valor del dígito correspondiente. Se va a tener una variable global que permita saber en qué operando se debe escribir. Una vez que se conoce si va en el primero o en el segundo, se le añade a la etiqueta de dicho operando el dígito correspondiente. Usando la forma abreviada de la instrucción de control de flujo, se chequea si la etiqueta tiene un cero para sustituirlo directamente o para añadirle el nuevo dígito. El código para los diez botones numéricos va a ser el siguiente:

```
private void but0_Click(object sender, EventArgs e)
{
    // Se añade el número al operando correspondiente
    if (iOperando==1)
    {
        // Se añade al primer operando
        if (lblOperando1.Text == "0"? "0": 
            lblOperando1.Text + "0");
    }
    else
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
// Se añade al segundo operando
lblOperando2.Text = lblOperando2.Text == "0" ? "0" :
    lblOperando2.Text + "0";
}

}

private void but1_Click(object sender, EventArgs e)
{
    // Se añade el número al operando correspondiente
    if (iOperando == 1)
    {
        // Se añade al primer operando
        lblOperando1.Text = lblOperando1.Text == "0" ? "1" :
            lblOperando1.Text + "1";
    }
    else
    {
        // Se añade al segundo operando
        lblOperando2.Text = lblOperando2.Text == "0" ? "1" :
            lblOperando2.Text + "1";
    }
}

private void but2_Click(object sender, EventArgs e)
{
    // Se añade el número al operando correspondiente
    if (iOperando == 1)
    {
        // Se añade al primer operando
        lblOperando1.Text = lblOperando1.Text == "0" ? "2" :
            lblOperando1.Text + "2";
    }
    else
    {
        // Se añade al segundo operando
        lblOperando2.Text = lblOperando2.Text == "0" ? "2" :
            lblOperando2.Text + "2";
    }
}

private void but3_Click(object sender, EventArgs e)
{
    // Se añade el número al operando correspondiente
    if (iOperando == 1)
    {
        // Se añade al primer operando
        lblOperando1.Text = lblOperando1.Text == "0" ? "3" :
            lblOperando1.Text + "3";
    }
}
```

7. Soluciones a los ejercicios

```
{  
    // Se añade al segundo operando  
    lb1Operando2.Text = lb1Operando2.Text == "0" ? "3" :  
        lb1Operando2.Text + "3";  
}  
}  
  
private void but4_Click(object sender, EventArgs e)  
{  
    // Se añade el número al operando correspondiente  
    if (iOperando == 1)  
    {  
        // Se añade al primer operando  
        lb1Operando1.Text = lb1Operando1.Text == "0" ? "4" :  
            lb1Operando1.Text + "4";  
    }  
    else  
    {  
        // Se añade al segundo operando  
        lb1Operando2.Text = lb1Operando2.Text == "0" ? "4" :  
            lb1Operando2.Text + "4";  
    }  
}  
  
private void but5_Click(object sender, EventArgs e)  
{  
    // Se añade el número al operando correspondiente  
    if (iOperando == 1)  
    {  
        // Se añade al primer operando  
        lb1Operando1.Text = lb1Operando1.Text == "0" ? "5" :  
            lb1Operando1.Text + "5";  
    }  
    else  
    {  
        // Se añade al segundo operando  
        lb1Operando2.Text = lb1Operando2.Text == "0" ? "5" :  
            lb1Operando2.Text + "5";  
    }  
}  
  
private void but6_Click(object sender, EventArgs e)  
{  
    // Se añade el número al operando correspondiente  
    if (iOperando == 1)  
    {  
        // Se añade al primer operando  
        lb1Operando1.Text = lb1Operando1.Text == "0" ? "6" :  
            lb1Operando1.Text + "6";  
    }  
}
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
else
{
    // Se añade al segundo operando
    lblOperando2.Text = lblOperando2.Text == "0" ? "6" :
        lblOperando2.Text + "6";
}
}

private void but7_Click(object sender, EventArgs e)
{
    // Se añade el número al operando correspondiente
    if (iOperando == 1)
    {
        // Se añade al primer operando
        lblOperando1.Text = lblOperando1.Text == "0" ? "7" :
            lblOperando1.Text + "7";
    }
    else
    {
        // Se añade al segundo operando
        lblOperando2.Text = lblOperando2.Text == "0" ? "7" :
            lblOperando2.Text + "7";
    }
}

private void but8_Click(object sender, EventArgs e)
{
    // Se añade el número al operando correspondiente
    if (iOperando == 1)
    {
        // Se añade al primer operando
        lblOperando1.Text = lblOperando1.Text == "0" ? "8" :
            lblOperando1.Text + "8";
    }
    else
    {
        // Se añade al segundo operando
        lblOperando2.Text = lblOperando2.Text == "0" ? "8" :
            lblOperando2.Text + "8";
    }
}

private void but9_Click(object sender, EventArgs e)
{
    // Se añade el número al operando correspondiente
    if (iOperando == 1)
```

7. Soluciones a los ejercicios

```
// Se añade al primer operando  
    1b1Operando1.Text = 1b1Operando1.Text == "0" ? "9" :  
        1b1Operando1.Text + "9";  
}  
else  
{  
    // Se añade al segundo operando  
    1b1Operando2.Text = 1b1Operando2.Text == "0" ? "9" :  
        1b1Operando2.Text + "9";  
}  
}  
}
```

El botón del punto decimal es similar a los numéricos, pero en este caso no es necesario chequear si hay un cero, puesto que el decimal sí que se añade en ese caso. Además, antes de terminar la pulsación se debe desactivar a sí mismo para que no se pueda volver a pulsar en el mismo operando.

```
private void butPunto_Click(object sender, EventArgs e)  
{  
    // Se añade el punto al operando correspondiente  
    if (iOperando == 1)  
    {  
        // Se añade al primer operando  
        1b1Operando1.Text = 1b1Operando1.Text + ",";  
    }  
    else  
    {  
        // Se añade al segundo operando  
        1b1Operando2.Text = 1b1Operando2.Text + ",";  
    }  
    // Se bloquea, porque sólo se puede poner una vez  
    butPunto.Enabled = false;  
}
```

El siguiente grupo de botones es el de las operaciones. Estos botones deben hacer también todos algo muy similar y lo único que variará será la variable correspondiente. Las acciones que van a realizar son:

- Escribir la operación correspondiente en la etiqueta a tal efecto.
- Cambiar al segundo operando.
- Activar de nuevo el botón del punto.
- Desactivar todos los botones de operación.

El código para estos botones es el siguiente:

```
private void butMas_Click(object sender, EventArgs e)
{
    // Se añade la operación
    lblOperacion.Text = "+";
    // Se cambia de operando
    iOperando = 2;
    // Se vuelve a activar el punto
    butPunto.Enabled = true;
    // Se bloquean las operaciones, porque sólo se puede
    // operar una vez
    butMas.Enabled = false;
    butMenos.Enabled = false;
    butPor.Enabled = false;
    butEntre.Enabled = false;
}

private void butMenos_Click(object sender, EventArgs e)
{
    // Se añade la operación
    lblOperacion.Text = "-";
    // Se cambia de operando
    iOperando = 2;
    // Se vuelve a activar el punto
    butPunto.Enabled = true;
    // Se bloquean las operaciones, porque sólo se puede
    // operar una vez
    butMas.Enabled = false;
    butMenos.Enabled = false;
    butPor.Enabled = false;
    butEntre.Enabled = false;
}

private void butPor_Click(object sender, EventArgs e)
{
    // Se añade la operación
    lblOperacion.Text = "*";
    // Se cambia de operando
    iOperando = 2;
    // Se vuelve a activar el punto
    butPunto.Enabled = true;
    // Se bloquean las operaciones, porque sólo se puede
    // operar una vez
    butMas.Enabled = false;
    butMenos.Enabled = false;
    butPor.Enabled = false;
    butEntre.Enabled = false;
}
```

7. Soluciones a los ejercicios

```
}

private void butEntre_Click(object sender, EventArgs e)
{
    // Se añade la operación
    lblOperacion.Text = "/";
    // Se cambia de operando
    iOperando = 2;
    // Se vuelve a activar el punto
    butPunto.Enabled = true;
    // Se bloquean las operaciones, porque sólo se puede
    // operar una vez
    butMas.Enabled = false;
    butMenos.Enabled = false;
    butPor.Enabled = false;
    butEntre.Enabled = false;
}
```

Y el botón que lleva la mayor complicación es el del igual, ya que debe calcular el resultado de la operación. Para saber qué operación se debe realizar basta con chequear la etiqueta de la operación, que tendrá guardado lo que se debe hacer.

A la hora de operar, se debe convertir en primer lugar cada operando a número doble y el resultado se convertirá otra vez a *String* para colocarlo en la etiqueta del resultado. Los métodos de conversión para ambos casos son *Convert.ToDouble* y *Convert.ToString*.

El código completo de este botón es el siguiente:

```
private void butIgual_Click(object sender, EventArgs e)
{
    // Se ejecuta la operación correspondiente
    switch (lblOperacion.Text)
    {
        case "+":
            lblResultado.Text = (Convert.ToDouble(lblOperando1.Text)
                + Convert.ToDouble(lblOperando2.Text)).To-
            String();
            break;
        case "-":
            lblResultado.Text = (Convert.ToDouble(lblOperando1.Text)
                - Convert.ToDouble(lblOperando2.Text)).To-
            String();
            break;
```

```
        case "*";
            lblResultado.Text = (Convert.ToDouble(lblOperando1.Text)
                * Convert.ToDouble(lblOperando2.Text)).ToString();
            break;
        case "/":
            lblResultado.Text = (Convert.ToDouble(lblOperando1.Text)
                / Convert.ToDouble(lblOperando2.Text)).ToString();
            break;
    }
}
```

El último botón es el que prepara todo para comenzar una nueva operación. Para ello se deben poner los valores iniciales de cada etiqueta, inicializar la variable del operando al primero y activar los botones que estaban desactivados. El código queda así:

```
private void butNueva_Click(object sender, EventArgs e)
{
    // Se reinicializan todos los elementos
    lblOperando1.Text = "0";
    lblOperando2.Text = "0";
    lblOperacion.Text = "";
    lblResultado.Text = "0";
    iOperando = 1;
    butPunto.Enabled = true;
    butMas.Enabled = true;
    butMenos.Enabled = true;
    butPor.Enabled = true;
    butEntre.Enabled = true;
}
```

Ejercicio 4.2

Se debe empezar realizando el diseño de la ventana. Hay que colocar los mismos objetos que aparecen en el diseño propuesto. Para ello se van a utilizar las siguientes herramientas:

- *Label*: etiquetas que contienen texto fijo.

7. Soluciones a los ejercicios

- *Button*: botones de comando.
- *GroupBox*: cajas agrupadas.
- *TrackBar*: barras de desplazamiento.
- *PictureBox*: control de imagen.

En el desarrollo del código, por un lado está la parte del color de fondo. En este caso se debe utilizar el método *Scroll* de cada una de las *Trackbars*, de manera que según se vaya moviendo cualquiera de ellas, ya se vaya modificando el color del fondo de la ventana. Para dicho cambio se debe modificar la propiedad *BackColor* de la ventana. Para calcular el color se debe usar la función *FromArgb* de la clase *Color* y generar la mezcla de valores rojo, verde y azul.

El código para cambiar el color es el siguiente:

```
e) private void traRojo_Scroll(object sender, EventArgs e)
{
    // Se cambia el color de la mezcla
    this.BackColor = Color.FromArgb(traRojo.Value,
        traVerde.Value, traAzul.Value);
}

private void traVerde_Scroll(object sender, EventArgs e)
{
    // Se cambia el color de la mezcla
    this.BackColor = Color.FromArgb(traRojo.Value,
        traVerde.Value, traAzul.Value);
}

private void traAzul_Scroll(object sender, EventArgs e)
{
    // Se cambia el color de la mezcla
    this.BackColor = Color.FromArgb(traRojo.Value,
        traVerde.Value, traAzul.Value);
}
```

Para la parte de recolocación de las imágenes, sólo se debe escribir un código en el evento *MouseDown* de la propia ventana. En los botones de opción no hay que crear ningún código, ya que funcionan solos y bastará con preguntar qué botón está seleccionado para saber la imagen que se debe mover de lugar.

En el evento *Mousedown* mencionado antes se recibe como argumento la variable *e*, que es del tipo *EventArgs*. Dicha variable tiene varias propiedades, aunque las que nos interesan en este caso son *X* e *Y*, que contendrán las coordenadas en las que se ha pulsado en la ventana. Una vez que se tengan las coordenadas, la función *Point* permitirá obtener un punto dentro de la ventana, el cual se puede asignar a la propiedad *Location* de las imágenes. Cuando se cambie su localización por la del punto en el que se ha pulsado el ratón, la imagen se recolocará en la ventana.

El código para la localización de las imágenes en la ventana es el siguiente:

```
private void frmEjercicio14_MouseDown(object sender,
    MouseEventArgs e)
{
    if (radCoche.Checked)
    {
        // Se coloca la imagen del coche en la ventana
        picImagen1.Location = new Point(e.X, e.Y);
    }
    else
    {
        if (radCamion.Checked)
        {
            // Se coloca la imagen del camión en la
            // ventana
            picImagen2.Location = new Point(e.X, e.Y);
        }
        else
        {
            // Se coloca la imagen de la moto en la
            // ventana
            picImagen3.Location = new Point(e.X, e.Y);
        }
    }
}
```

El último paso es el código del botón para salir de la aplicación, que será como el que se ha visto en los otros ejemplos y ejercicios:

```
private void butSalir_Click(object sender, EventArgs e)
{
    // Salir de la aplicación
    this.Close();
}
```

Ejercicio 4.3

Se empieza realizando el diseño de la ventana y colocando los mismos objetos que aparecen en el diseño propuesto. Para ello se van a utilizar las siguientes herramientas:

- *Label*: etiquetas que contienen texto fijo.
- *Button*: botones de comando.
- *ComboBox*: cajas combinadas.
- *ListBox*: listas de elementos.

También se va a diseñar la base de datos con las dos tablas necesarias para la aplicación. Se hará en *Access* y las tablas tendrán que ser como las siguientes:

Películas		
	Field Name	Data Type
	Código	AutoNumber
	Título	Text

Actores		
	Field Name	Data Type
	Código	AutoNumber
	Nombre	Text
	Apellido	Text
	Película	Number

Para desarrollar el código, se va a empezar por la parte de la conexión con la base de datos. Aparte de la variable general para la conexión, se van a usar las rutinas para abrir y cerrar dicha conexión. El código debe ser como el siguiente:

```
// Variables principales
OleDbConnection conBD = new OleDbConnection();

public void abrirConexion()
{
    // Se crea el string de conexión a la base de datos
    conBD.ConnectionString = @"Provider=Microsoft.ACE.
        OLEDB.12.0; Data Source=.\\BD\\Grabaciones.accdb;Per-
        sist Security Info=False";
    try
    {
        // Se abre la base de datos
        conBD.Open();
    }
    catch (Exception ex)
    {
        MessageBox.Show("Error de conexión" + ex);
    }
}

public void cerrarConexion()
{
    // Se chequea si está la conexión abierta
    if (conBD.State == ConnectionState.Open)
    {
        // Se cierra la conexión
        conBD.Close();
    }
}
```

Cuando arranque la aplicación, hay que establecer la conexión con la base de datos y, acto seguido, llenar la lista con los títulos de las películas. Para ello hay que realizar la consulta a la tabla de películas y conviene ordenar los resultados para que sea más sencillo después encontrar el título deseado. Las funciones para inicializar la aplicación van a ser las siguientes:

```
private void frmEjercicio19_Load(object sender, EventArgs e)
{
    // Se abre la conexión con la base de datos
    abrirConexion();
    // Se inicializa la lista de películas
    leerPeliculas();
```

7. Soluciones a los ejercicios

```
}

private void leerPeliculas()
{
    try
    {
        // Se vacía la lista
        cboPeliculas.Items.Clear();
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM Peliculas "
            + "ORDER BY Título";
        OleDbCommand oleComando = new OleDbCommand(sConsulta,
conBD);
        OleDbDataReader oleReader = oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno
        while (oleReader.Read())
        {
            // Se añade la película a la lista
            cboPeliculas.Items.Add(
                Convert.ToString(oleReader["Título"]));
        }
        // Se coloca el primero como principal
        cboPeliculas.Text = cboPeliculas.Items[0].ToString();
    }
    catch (Exception ex)
    {
        // Si salta una excepción, se muestra el error
        MessageBox.Show("Error: " + ex);
    }
}
```

Una vez que se obtenga el listado de las películas, lo siguiente que hay que hacer es controlar el título de la película seleccionada para actualizar la lista de actores y actrices. Para poder controlar el cambio de selección se usa el evento *SelectedIndexChanged* en la caja combinada. Cuando se produzca dicho evento se llamará a una función que lea la tabla de actores de la base de datos y muestre la información en la otra lista.

En la función de lectura hay que comenzar vaciando la lista. Después se realiza la consulta a la base de datos incluyendo las dos tablas, puesto que lo que se conoce es el título, que únicamente se encuentra en la tabla de películas. A la hora de poner las condiciones, se indica que el código debe coincidir en ambas tablas y que el título debe ser el que esté seleccionado en ese momento. También conviene establecer un orden para los datos que se van a mostrar.

Con las respuestas obtenidas en la consulta se hace un bucle al que se vayan añadiendo los elementos a la lista con las conversiones a texto correspondientes. El evento que hay que tratar y la función asociada son los siguientes:

```
private void cboPeliculas_SelectedIndexChanged(object sender, EventArgs e)
{
    // Se actualiza la lista de actores y actrices
    leerActores();
}

private void leerActores()
{
    try
    {
        // Se vacía la lista
        lstActores.Items.Clear();
        // Se realiza la consulta
        String sConsulta = "SELECT * "
            + "FROM Actores, Peliculas "
            + "WHERE Pelicula=Peliculas.Codigo "
            + "AND Título='" + cboPeliculas.Text + "' "
            + "ORDER BY Nombre, Apellido";
        OleDbCommand oleComando = new OleDbCommand(sConsulta,
            conBD);
        OleDbDataReader oleReader =
            oleComando.ExecuteReader();
        // Se lee el resultado para ver si hay alguno
        while (oleReader.Read())
        {
            // Se añade la película a la lista
            lstActores.Items.Add(Convert.ToString(
                oleReader["Nombre"] + " "
                + Convert.ToString(oleReader["Apellido"])));
        }
    }
    catch (Exception ex)
    {
        // Si salta una excepción, se muestra el error
        MessageBox.Show("Error: " + ex);
    }
}
```

Por último, el código para el botón de salida es simplemente el que reabre la llamada a la función del cierre de la conexión y la instrucción que

7. Soluciones a los ejercicios

```
private void butSalir_Click(object sender, EventArgs e)
{
    // Se cierra la conexión antes de terminar
    cerrarConexion();
    // Finaliza el programa
    this.Close();
}
```

Ejercicio 4.4

En este ejercicio, todos los eventos que se van a programar están en las cajas de texto en el momento en que se modifica su contenido. Para ello se va a utilizar el evento *TextChanged* de dichas cajas.

Como todas las cajas de texto van a realizar los mismos cálculos, lo mejor es realizar una función que calcule todo lo necesario y que esa función sea llamada desde cada una de las cajas de texto. En el siguiente código se pueden ver estas llamadas:

```
private void txtEquipo11_TextChanged(object sender, EventArgs e)
{
    CalcularMedias();
}

private void txtEquipo21_TextChanged(object sender, EventArgs e)
{
    CalcularMedias();
}

private void txtEquipo12_TextChanged(object sender, EventArgs e)
{
    CalcularMedias();
}

private void txtEquipo22_TextChanged(object sender, EventArgs e)
{
    CalcularMedias();
}

private void txtEquipo13_TextChanged(object sender, EventArgs e)
{
    CalcularMedias();
}

private void txtEquipo23_TextChanged(object sender, EventArgs e)
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
{  
    CalcularMedias();  
}  
  
private void txtEquipo14_TextChanged(object sender, EventArgs e)  
{  
    CalcularMedias();  
}  
  
private void txtEquipo24_TextChanged(object sender, EventArgs e)  
{  
    CalcularMedias();  
}  
  
private void txtEquipo15_TextChanged(object sender, EventArgs e)  
{  
    CalcularMedias();  
}  
  
private void txtEquipo25_TextChanged(object sender, EventArgs e)  
{  
    CalcularMedias();  
}
```

La función para el cálculo debe ser como la que se muestra a continuación.
Está bien comentada para que se aprecie bien cada apartado.

```
private void CalcularMedias()  
{  
    // Primero se obtienen los valores para el cálculo  
    int iValor11 = 0;  
    int iValor12 = 0;  
    int iValor13 = 0;  
    int iValor14 = 0;  
    int iValor15 = 0;  
    int iValor21 = 0;  
    int iValor22 = 0;  
    int iValor23 = 0;  
    int iValor24 = 0;  
    int iValor25 = 0;  
    try  
    {  
        // Se obtiene el valor de la caja de texto  
        // correspondiente  
        iValor11 = Convert.ToInt16(txtEquipo11.Text);  
    }  
    catch (Exception ex)  
    {  
        // Se maneja el error  
    }  
}
```

7. Soluciones a los ejercicios

```
txtEquipo11.Text = "0";
iValor11 = 0;
}
try
{
    // Se obtiene el valor de la caja de texto
    // correspondiente
    iValor12 = Convert.ToInt16(txtEquipo12.Text);
}
catch (Exception ex)
{
    // Si el valor ha saltado una excepción, se pone a 0
    txtEquipo12.Text = "0";
    iValor12 = 0;
}
try
{
    // Se obtiene el valor de la caja de texto
    // correspondiente
    iValor13 = Convert.ToInt16(txtEquipo13.Text);
}
catch (Exception ex)
{
    // Si el valor ha saltado una excepción, se pone a 0
    txtEquipo13.Text = "0";
    iValor13 = 0;
}
try
{
    // Se obtiene el valor de la caja de texto
    // correspondiente
    iValor14 = Convert.ToInt16(txtEquipo14.Text);
}
catch (Exception ex)
{
    // Si el valor ha saltado una excepción, se pone a 0
    txtEquipo14.Text = "0";
    iValor14 = 0;
}
try
{
    // Se obtiene el valor de la caja de texto
    // correspondiente
    iValor15 = Convert.ToInt16(txtEquipo15.Text);
}
catch (Exception ex)
{
    // Si el valor ha saltado una excepción, se pone a 0
    txtEquipo15.Text = "0";
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
}

try
{
    // Se obtiene el valor de la caja de texto
    // correspondiente
    iValor21 = Convert.ToInt16(txtEquipo21.Text);
}
catch (Exception ex)
{
    // Si el valor ha saltado una excepción, se pone a 0
    txtEquipo21.Text = "0";
    iValor21 = 0;
}

try
{
    // Se obtiene el valor de la caja de texto
    // correspondiente
    iValor22 = Convert.ToInt16(txtEquipo22.Text);
}
catch (Exception ex)
{
    // Si el valor ha saltado una excepción, se pone a 0
    txtEquipo22.Text = "0";
    iValor22 = 0;
}

try
{
    // Se obtiene el valor de la caja de texto
    // correspondiente           iValor23 = Convert.
ToInt16(txtEquipo23.Text);
}
catch (Exception ex)
{
    // Si el valor ha saltado una excepción, se pone a 0
    txtEquipo23.Text = "0";
    iValor23 = 0;
}

try
{
    // Se obtiene el valor de la caja de texto
    // correspondiente
    iValor24 = Convert.ToInt16(txtEquipo24.Text);
}
catch (Exception ex)
{
    // Si el valor ha saltado una excepción, se pone a 0
    txtEquipo24.Text = "0";
    iValor24 = 0;
}
```

7. Soluciones a los ejercicios

```
{  
    // Se obtiene el valor de la caja de texto  
    // correspondiente  
    iValor25 = Convert.ToInt16(txtEquipo25.Text);  
}  
catch (Exception ex)  
{  
    // Si el valor ha saltado una excepción, se pone a 0  
    txtEquipo25.Text = "0";  
    iValor25 = 0;  
}  
// Se calculan primero los totales para cada partido  
lblTotal1.Text = Convert.ToString(iValor11 + iValor21);  
lblTotal2.Text = Convert.ToString(iValor12 + iValor22);  
lblTotal3.Text = Convert.ToString(iValor13 + iValor23);  
lblTotal4.Text = Convert.ToString(iValor14 + iValor24);  
lblTotal5.Text = Convert.ToString(iValor15 + iValor25);  
// Después se calculan los totales ponderados  
lblTotalPonderado1.Text = lblTotal1.Text;  
lblTotalPonderado2.Text =  
    Convert.ToString(Convert.ToInt16(lblTotal2.Text) *  
1.1);  
lblTotalPonderado3.Text =  
    Convert.ToString(Convert.ToInt16(lblTotal3.Text) *  
1.2);  
lblTotalPonderado4.Text =  
    Convert.ToString(Convert.ToInt16(lblTotal4.Text) *  
1.3);  
lblTotalPonderado5.Text =  
    Convert.ToString(Convert.ToInt16(lblTotal5.Text) *  
1.4);  
// Por ultimo, se calculan las medias globales  
lblMediaEquipo1.Text =  
    Convert.ToString((Convert.ToDouble(txtEquipo11.Text)  
+ (Convert.ToDouble(txtEquipo12.Text) * 1.1)  
+ (Convert.ToDouble(txtEquipo13.Text) * 1.2)  
+ (Convert.ToDouble(txtEquipo14.Text) * 1.3)  
+ (Convert.ToDouble(txtEquipo15.Text) * 1.4)) / 6);  
if (lblMediaEquipo1.Text.IndexOf(",") > 1)  
{  
    lblMediaEquipo1.Text = lblMediaEquipo1.Text.Sub-  
string(0,  
        lblMediaEquipo1.Text.IndexOf(",") );  
}  
lblMediaEquipo2.Text =  
    Convert.ToString((Convert.ToDouble(txtEquipo21.Text)  
+ (Convert.ToDouble(txtEquipo22.Text) * 1.1)  
+ (Convert.ToDouble(txtEquipo23.Text) * 1.2)  
+ (Convert.ToDouble(txtEquipo24.Text) * 1.3))
```

Desarrollo de aplicaciones C# con Visual Studio .NET

```
if (lblMediaEquipo2.Text.IndexOf(",") > 1)
{
    lblMediaEquipo2.Text = lblMediaEquipo2.Text.Substring(0,
        lblMediaEquipo2.Text.IndexOf(","));
}
lblMediaTotalPonderada.Text =
    Convert.ToString((Convert.ToDouble(lblTotalPonderado1.Text)
    + Convert.ToDouble(lblTotalPonderado2.Text)
    + Convert.ToDouble(lblTotalPonderado3.Text)
    + Convert.ToDouble(lblTotalPonderado4.Text)
    + Convert.ToDouble(lblTotalPonderado5.Text)) / 6);
if (lblMediaTotalPonderada.Text.IndexOf(",") > 1)
{
    lblMediaTotal.Text = lblMediaTotalPonderada.Text.Substring(0,
        lblMediaTotalPonderada.Text.IndexOf(","));
}
else
{
    lblMediaTotal.Text = lblMediaTotalPonderada.Text;
}
// Se ponen los colores finales
if (lblMediaEquipo1.Text == lblMediaEquipo2.Text)
{
    // Se pone el color negro
    lblMediaEquipo1.ForeColor =
        System.Drawing.Color.FromArgb(0, 0, 0);
    lblMediaEquipo2.ForeColor =
        System.Drawing.Color.FromArgb(0, 0, 0);
}
else
{
    if (Convert.ToInt16(lblMediaEquipo1.Text) >
        Convert.ToInt16(lblMediaEquipo2.Text))
    {
        // Se ponen los colores verde y rojo
        lblMediaEquipo1.ForeColor =
            System.Drawing.Color.FromArgb(0, 255, 0);
        lblMediaEquipo2.ForeColor =
            System.Drawing.Color.FromArgb(255, 0, 0);
    }
    else
    {
        // Se ponen los colores rojo y verde
        lblMediaEquipo1.ForeColor =
            System.Drawing.Color.FromArgb(255, 0, 0);
        lblMediaEquipo2.ForeColor =
            System.Drawing.Color.FromArgb(0, 255, 0);
    }
}
```

7. Soluciones a los ejercicios

El resultado al ejecutar la aplicación debe ser como el que se muestra en el ejemplo de esta imagen:

The screenshot shows a Windows application window titled "Borja Orbegozo". The main title bar has the name "Borja Orbegozo" and standard window controls (minimize, maximize, close). Below the title bar is a sub-header "Predicción de resultados para partidos de baloncesto". The main content is a table with the following data:

	<u>Equipo 1</u>	<u>Equipo 2</u>	<u>Total</u>	<u>Total ponderado</u>
Resultado partido 1:	50	60	110	110
Resultado partido 2:	60	40	100	110
Resultado partido 3:	70	90	160	192
Resultado partido 4:	80	50	130	169
Resultado partido 5:	90	120	210	294
Medias:	71	74	145	145,8333

At the bottom center of the window is a button labeled "Salir".