Capítulo 3

PROGRAMACION ORIENTADA A OBJETOS CON C#

El paradigma de programación orientada a objetos ha sido el estilo de programación que ha permitido revolucionar el desarrollo de software en la informática moderna. A pesar de que se afirma que cualquier programa desarrollado bajo el modelo de programación orientada a objetos, también se hubiera podido desarrollar con el paradigma de programación estructurada, la verdad es que las cosas que se han logrado en materia de software son más fáciles de concebir, diseñar y programar con la noción de objetos que con la programación netamente estructurada. La orientación a objetos es un modelo que utiliza nociones del mundo real, en el cual vivimos los seres humanos, donde encontramos objetos por todas partes, los cuales se relacionan, interactúan y se asocian para permitir la solución de un problema, se pueden manipular individualmente o en conjunto, e incluso muchos de ellos pueden poseer o adquirir comportamiento inteligente.

En esencia, la programación orientada a objetos nos es más que un modelo de programación donde un programa es dividido en módulos de software independientes unos de otros pero con capacidad de interactuar entre si para cumplir con un objetivo. Dichos módulos están conformados básicamente por datos y funciones que se encargan de manipularlos internamente para mostrar al exterior únicamente los resultados que necesita el programa que los va a utilizar. En el desarrollo de este capítulo vamos a describir las nociones básicas de la programación orientada a objetos y la forma como C# las implementa.

La mayoría de nosotros iniciamos nuestro aprendizaje de la programación de computadores en el modelo estructurado, por lo cual a más de uno nos cuesta la adaptación al modelo orientado a objetos. Esta situación ha hecho que mucha de la literatura sobre programación orientada a objetos utilice ejemplos que, aunque muy didácticos, pueden hacer más complicado el salto a la tarea práctica de programar con objetos reales de software. En muchas explicaciones y literatura sobre el tema hemos escuchado o leído sobre la clase *felino* y todos los grupos familiares que de ella se pueden derivar: *gatos*, *tigres*, *leones*, *linces*, *la Pantera Rosa*, etc. Pero en la práctica muy pocas de nuestras aplicaciones, y más aún las que se ejecutarán sobre .NET, se las tienen que ver con este tipo de especimenes y más bien lo que encontraremos son objetos como: ventanas, cajas de texto, botones, menús, números, puntos, bases de datos, puertos de comunicaciones, entre muchos otros. Y es sobre estos objetos que vamos a centrar nuestro trabajo, evitando en lo posible las explicaciones con objetos que poco o nada tienen que ver con programas reales de software.

El concepto fundamental de la programación orientada a objetos es la *clase* y existen muchas de ellas implementadas en .NET, y están listas para permitirnos crear la gran mayoría de objetos que podemos llegar a necesitar en un programa de consola, tipo Windows o tipo web. Aquellos objetos para los cuales no existan clases en .NET podemos crearlas utilizando cualquier lenguaje de programación y un compilador que cumplan la especificación para ejecutarse sobre el *framework .NET*. C# es uno de esos muchos lenguajes, pero por ahora es el único que ha sido diseñado y desarrollado entera y exclusivamente para este entorno, y todos aquellos que cumplan con la misma especificación .NET.

Clases

La *clase* es el elemento fundamental de la programación orientada a objetos con el lenguaje C#. Aunque, dentro de este modelo de programación, existen muchas definiciones válidas para el concepto de clase, en la práctica una clase no es más que una plantilla de software que sirve para construir cualquier cantidad de objetos. Por ejemplo, en .NET existe una clase llamada *Form* que sirve como molde para construir cualquier ventana que necesite una aplicación tipo Windows. Las ventanas que nosotros observamos en la pantalla de nuestro computador, son los objetos generados con esa plantilla. Pueden haber muchos objetos generados con esa plantilla, pero lo que los puede hacer diferentes a unos de otros, son los valores que se asignan a sus atributos como: alto, ancho, color, título, etc.

Desde la perspectiva del lenguaje de programación, *una clase es un tipo*, que al igual que los tipos estándar, sirve para declarar variables cuya estructura es una fiel copia de ella. Estas variables reciben el nombre de *objetos* y son los elementos que manipula el programador para desarrollar su programa. En la evolución del desarrollo de software y los lenguajes de programación, es posible que en el futuro tan solo se hable de tipos, en vez de clases. Serán tantas las clases que cualquier programador podrá encontrar en las redes de comunicaciones, como Internet, que más que preocuparse por construir un molde deberá poner todo su potencial en manipular los objetos que de él se puedan obtener, y todo se verá con la simpleza que hoy se mira a los tipos de datos como entero, cadena de texto, booleano o byte, entre otros.

Pero, por ahora volvamos a nuestra realidad que nos exige a nosotros definir los tipos o clases que el entorno de desarrollo no haya definido, o que siendo tantos, no sepamos que ya existen, y que los necesitemos para nuestros programas. En C# una clase se define mediante la palabra clave ***class*** y una sintaxis básica que encontramos para definirla, es la siguiente:

```
class NombreClase
{
      // Miembros
}
```

Por ejemplo, supongamos que deseamos un componente de software para procesos matemáticos que nos permitan manipular números complejos[1]. La clase que nos permitirá procesar estos números la llamaremos *Complejo*, y se puede definir como,

```
class Complejo
{
      //Miembros de la clase Complejo
}
```

Sin embargo, las clases deben cumplir unos niveles de seguridad que exigen el manejo del control de accesibilidad a ellas, sobre todo por parte de agentes externos al proyecto de software donde se hayan definido. Es por esto que la definición de cualquier clase debe ir antecedida de una palabra clave que determina la accesibilidad que admite dicha clase. La sintaxis C# para definir una clase es la siguiente:

```
[public | private | protected | internal]
class NombreClase
{
      // Miembros
}
```

[1] Los números complejos son un conjunto de números que amplían las operaciones de las matemáticas al plano cartesiano. Cada número complejo se puede interpretar como un punto o un vector en el plano XY.

La sección entre corchetes, que indica las palabras clave de accesibilidad, *public*, *private*, *protected*, *internal*, que pueden utilizarse en la definición de una clase, es opcional e indica cual es el nivel de acceso que se va a permitir sobre la clase. Si no se especifica ningún nivel de accesibilidad, el compilador la define por defecto como *internal*, lo cual significa que solo se permite el acceso a las clases que hacen parte del mismo ensamblado. Mediante las palabras de accesibilidad se pueden especificar los siguientes cinco niveles de protección para una clase:

Acceso	Seguridad
`public`	No existe ninguna restricción de acceso.
`protected`	Solo pueden tener acceso la clase contenedora o los tipos derivados de esta clase.
`internal`	Únicamente se permite el acceso al ensamblado actual.
`protected internal`	El acceso está limitado al ensamblado actual o los tipos derivados de la clase contenedora.
`private`	Solo se permite el acceso al tipo contenedor.

Para nombrar una clase se sugiere utilizar una cadena de caracteres que inicia con una letra mayúscula y cuyo significado es familiar para el programador. Aquí es muy importante tener en cuanta cual es la funcionalidad básica de la clase, para darle un nombre que resulte significativo y fácil de recordar posteriormente. Podemos tener clases como: Complejo, Estudiante, DocumentoImpresion, RedNeuronal, ConexionBaseDatos.

Objetos

Cuando se diseña y programa una aplicación de software con el modelo de programación orientada a objetos, lo que se hace en la práctica es construir un conjunto de plantillas de objetos, o lo que se conoce como clases, las cuales permiten definir variables en memoria que se conocen con el nombre de *objetos* o *instancias*[1] de clase. Desde esta perspectiva, el término *clase* gana mayor significado, ya que se puede definir como una plantilla que permite generar una *clase de objetos*.

Para crear un objeto de una determinada clase, se procede básicamente en dos pasos: primero se declara una variable con el tipo que representa la clase y luego se le asigna memoria con el operador ***new***. Por ejemplo, tomando la clase *Complejo* definida anteriormente, la siguiente línea crea un objeto a partir de esta plantilla:

```
Complejo z;
```

Cuando el programa ejecuta esta línea, solo posee una dirección de memoria donde se inicia la estructura de la variable `z`. Para asignarle la memoria total conforme al tamaño de la estructura que describe la clase, es necesario aplicar el operador *new*, así:

```
z = new Complejo();
```

[1] Muchos autores utilizan la palabra *instancia* para referirse a la variable declarada con el tipo que representa la clase. Este término se ha originado como producto de la traducción directa de la palabra *instance* que se utiliza en inglés, donde quiere decir *representación concreta y particular de una clase*. En general da lo mismo hablar de *objeto* que de *instancia*.

A partir de aquí, el sistema conoce con exactitud la ubicación y tamaño ocupado en memoria por la variable z y estará en condiciones de colocar en el sitio que corresponda cada uno de los datos y demás elementos que conforman el objeto.

En C#, tanto la declaración como la asignación de memoria para un objeto puede hacerse en una sola línea de código, como se muestra enseguida para nuestro ejemplo:

```
Complejo z = new Complejo();
```

Elementos de una clase

En la teoría general de programación orientada a objetos, las clases están conformadas por miembros, los cuales se clasifican básicamente en *atributos* y *métodos*. En vista de la complejidad que han ido adquiriendo los programas de computador y las posibilidades que los lenguajes de programación tratan de poner a disposición del programador, también ha sido necesario realizar una clasificación más detallada de los miembros que componen a una clase. Para realizar una identificación más precisa de los elementos de una clase en C# estos miembros los podemos considerar clasificados en:

Constructores
Destructores
Constantes
Campos
Métodos
Propiedades
Indizadores
Operadores
Eventos
Delegados
Estructuras

Aunque en la teoría general de programación orientada a objetos se habla de *atributos* para referirse a las variables que se encargan de almacenar los datos específicos de un objeto, en la especificación .NET este término tiene una connotación diferente, y en vez de eso se hablará de *campos*. Los campos son los elementos o variables internas que sirven para almacenar los datos que dan funcionalidad a los objetos derivados de una determinada clase. Corresponden a variables de cualquier tipo, ya sea de los tipos estándar del lenguaje o tipos definidos por el programador. En general, son campos todas las variables de cualquier tipo declaradas en el cuerpo de la clase (por fuera de cualquier método o función) que permiten guardar los datos básicos de un objeto.

Retomando nuestra clase *Complejo*, los campos básicos que podríamos inicialmente definir para este tipo, serían variables numéricas para almacenar la parte real y la parte imaginaria de un número complejo. Recordemos que los complejos son números que tienen la forma matemática,

$$a + bi$$

donde a y b son números reales denominados *parte real* y *parte imaginaria*, respectivamente, e i es un símbolo matemático que sirve para representar a $\sqrt{-1}$. En este punto todo indica que la clase Complejo debe estar conformada por dos variables básicas que van a permitir guardar el valor de la parte real y la parte imaginaria. A estas variables se les llamará campos.

Entonces la clase complejo se podría definir como,

```
class Complejo
{
      public double real;
      public double imaginario;
}
```

En adelante con la plantilla *Complejo* podremos definir variables (objetos) que permiten manipular números que, a diferencia de los tradicionales, como los enteros y los reales, constan de dos valores numéricos. Estos valores, aunque independientes uno de otro, conforman una estructura matemática sólida que permite solucionar muchos problemas relacionados con el manejo de puntos y vectores en el plano, tratando a estos elementos como si fueran simples números.

Volviendo a nuestro tema de interés, podemos decir que esta es una clase perfectamente válida, aunque muy limitada, ya que solo puede definir objetos para guardar y leer las partes de un número complejo. Los campos `real` e `imaginario` se han declarado como públicos para permitir ser escritos y leídos directamente desde el exterior de la clase. Esta es una forma de definir una clase que un buen programador jamás debería utilizar, por que implica dejar los datos desprotegidos ante el mundo exterior, algo inadmisible dentro de las reglas de diseño y programación orientada a objetos. Además, no siempre se va a requerir que los campos puedan ser leídos y escritos desde afuera de la clase, a veces es necesario que un campo sea de solo lectura o de solo escritura. Como regla general, los campos siempre se deben definir como privados y la comunicación con el mundo exterior se debe realizar a través de las *propiedades*, a las cuales no referiremos en las siguientes secciones.

Toda clase debe disponer de medios que le permitan a toda costa proteger sus datos, para evitar posibles daños provenientes del mundo exterior. Esto se llama *encapsulamiento*, uno de los principios fundamentales de la programación orientada a objetos. Pero, por ahora en aras de iniciar con pie firme la comprensión de estos nuevos conceptos, vamos a desarrollar el primer ejemplo de programación orientada a objetos en C#, propiamente dicha, utilizando esta sencilla clase.

Ejemplo 3.1 Complejos, elementales

El programa que vamos desarrollar a continuación implementa la clase *Complejo* para luego utilizarla en la definición de dos objetos que se encargarán de capturar los datos de dos números complejos y mostrarlos en pantalla.

```
/* Archivo: Ejemplo31.cs  */

using System;
class Complejo
{
      // Campos
      public double real;
      public double imaginario;
}

public class NumerosComplejos
{
      static void Main()
      {
            // Declarar dos objeto p y q de la clase Complejo
            Complejo p;
            Complejo q;
```

```
            // Asignar memoria a los objetos
            p = new Complejo();
            q = new Complejo();
            // Cargar los datos del número complejo p
            p.real = 4;
            p.imaginario = 10;
            // Cargar los datos del número complejo q
            q.real = -4;
            q.imaginario = 1;
            // Mostrar los números complejos p y q
            Console.WriteLine("p = {0} + {1}i", p.real, p.imaginario);
            Console.WriteLine("q = {0} + {1}i", q.real, q.imaginario);
      }
}
```

Compile este archivo, con la instrucción

```
> csc ejemplo31.cs
```

El ejemplo nos ha mostrado la forma de crear en C# un tipo llamado *Complejo*, que aunque no cumple con las reglas básicas del diseño orientado a objetos, no presenta ningún problema para el compilador. La pobreza de nuestra clase *Complejo*, se observa con mayor detalle en la parte final del programa, cuando se desea mostrar los valores de los números que almacena cada objeto, y es necesario recurrir dos veces al formateo de la salida mediante el método *WriteLine* de la clase *Console*. Este hecho debería haber sido resuelto por la propia clase, quién debería ser capaz de realizar la mayoría de tareas en forma autónoma sin tener que recurrir a elementos externos. Esta capacidad de no depender de recursos externos en el desarrollo de las tareas propias de un objeto, se conoce con el nombre de *modularidad*, otra característica importante de la programación orientada a objetos.

Elementos básicos de la orientación a objetos

Antes de continuar con la descripción de los componentes de una clase en C#, y su correspondiente uso en la programación, vamos a analizar aquellas características fundamentales que identifican a cualquier lenguaje orientado a objetos y la forma como C# las implementa. Aunque la descripción en detalle de estos conceptos es demasiado amplia, en este apartado no se pretende hacer un desarrollo en profundidad de los mismos, sino únicamente dar una aproximación de los mismos a nuestro lenguaje de interés. Igual, muchos de esos elementos se asimilan mejor con la práctica que mediante extensas descripciones teóricas.

Aunque aun hoy en día no existe un acuerdo pleno, entre los teóricos de la ingeniería de software, sobre cuales deben ser los elementos fundamentales que caractericen a un verdadero lenguaje orientado a objetos. En la mayoría de teorías sobre el tema se identifican los siguientes:

- Abstracción
- Encapsulamiento
- Modularidad
- Herencia
- Polimorfismo

La forma como uno u otro lenguaje de programación implementa cada una de estas propiedades, es lo que se coloca en la mesa de discusión. Muchos puritanos del paradigma orientado a objetos no ven con buenos ojos los trucos que algunos lenguajes

de programación utilizan para asegurar el soporte a alguna de estas propiedades. Un ejemplo, de este desacuerdo, es lo que sucede con C++. Este lenguaje, puede soportar la programación con clases y sin ellas, o incluso haciendo una combinación de los dos métodos. De hecho C++ se considera tan solo una ampliación del lenguaje C, un digno representante de la programación estructurada. Este aspecto hace que para muchos C++ no sea digno de formar parte de los lenguajes orientados a objetos.

El lenguaje C# ha sido aceptado en la familia orientada a objetos, pero para algunos, se identifican en él algunas debilidades que no lo hacen el más digno representante de esta estirpe. De todas formas, en este mundo de la programación cada programador defiende lo que mejor conoce, e incluso muchos nos hemos dejado llevar por aquello que impone la moda más que por la practicidad que nos pueda ofrecer uno u otro lenguaje de programación.

Más que unas cuantas líneas sobre los elementos de la programación orientada a objetos, lo que realmente nos dirá que tan fuerte es un lenguaje como C# en su implementación, será la práctica en la programación de verdaderos proyectos de software quien nos muestre sus debilidades y fortalezas.

A continuación se describe en forma básica el concepto que encierran estas propiedades y la forma como C# trabaja para lograr su cumplimiento.

Abstracción

En términos generales se entiende por *abstracción* la posibilidad de visualizar únicamente aquellos aspectos que interesen de un todo, dejando de lado los detalles que, aunque importantes, no son de interés para un determinado propósito. Por ejemplo, el ratón de nuestro computador es el resultado de la abstracción de todo un sistema de hardware y software que permite realizar algunas acciones sobre un programa de computador. La ingeniería aplicada a este mecanismo ha permitido alcanzar un nivel de abstracción tal, que para los usuarios lo único que interesa son los efectos visualizados en la pantalla del computador como consecuencia de los movimientos y pulsaciones de los botones de este dispositivo. Se dirá entonces, que el ratón es el resultado de la abstracción de un complejo mecanismo que permite interactuar con la interfaz gráfica de un programa de computador.

En el campo de la programación de computadores se debe entender una abstracción como una visión externa de un elemento, en el cual no interesa aquello que va por dentro. Es evidente que esta forma de ver las cosas no es un tema nuevo dentro de la programación, pues en la misma programación procedimental las funciones que crea el programador, o aquellas que ya vienen precompiladas con el sistema operativo, ya manejan un nivel de abstracción que nos permite despreocuparnos de aquellos detalles que la hacen funcionar y centrarnos únicamente en la forma como se utilizan esos elementos y los resultados que nos pueden devolver.

El elemento fundamental de la programación orientada a objetos es la clase, y es precisamente este elemento una abstracción que determina el comportamiento y funcionalidad de un conjunto de elementos de software que cumplen unas características que los hacen semejantes unos a otros. Es obvio que un programador tendrá que preocuparse, sobre todo en el desarrollo de sus propias clases, de los detalles que hacen funcionar correctamente a los objetos que se derivan de ellas, pero el éxito de su trabajo consiste precisamente en lograr un alto nivel de abstracción que le permita asegurar que cuando se vayan a utilizar algunos de su componentes desarrollados, ya sea por él mismo o un tercero, no sea importante recordar como funciona cada cosa por dentro.

La capacidad de lograr buenos niveles de abstracción es el pilar fundamental de un buen lenguaje orientado a objetos como C#. Pero la forma como se logra una buena

abstracción no solo depende del lenguaje de programación sino también de la creatividad y capacidad del programador para identificar y abstraer aquellos elementos que resulten fundamentales y relevantes en determinadas situaciones.

Para mostrar un buen ejemplo sobre abstracción demos un vistazo a los programas que interactúan con el usuario mediante interfaces gráficas que conocemos con el nombre de *ventanas*. Un entorno de desarrollo como .NET pone a disposición del programador una o varias clases que permiten definir este tipo de objetos. El nivel de abstracción que aplica .NET es tan elevado, que muchas de las funcionalidades de estas ventanas ya se encuentran incluidas en la clase que las identifica. Cualquier ventana ya esta lista para responder a algunas acciones del ratón, especialmente cuando se hace clic sobre sus botones de control y cuando se la arrastra desde la barra de títulos. El programador no necesita conocer como funciona cada cosa, sino únicamente la forma como se deben configurar algunos de sus elementos para lograr el efecto deseado.

Una ventana estándar en cualquier entorno gráfico, como Windows, tiene una forma rectangular y la caracterizan una barra de títulos, un icono que esconde un menú desplegable y tres botones, maximizar, restaurar y cerrar. El proceso para crear esa ventana involucra una serie de complejas acciones que incluyen desde elementos matemáticos hasta principios electrónicos sobre procesamiento digital de gráficos. Pero, en las clases que ofrece .NET, todo eso ha sido aislado del interés del programador, y solo se pone a su disposición unas cuantas propiedades y métodos que le permiten modificar cualquier aspecto de la ventana sin tener que preocuparse de los detalles internos. En otras palabras se ha realizado una abstracción del mecanismo que permite construir una ventana.

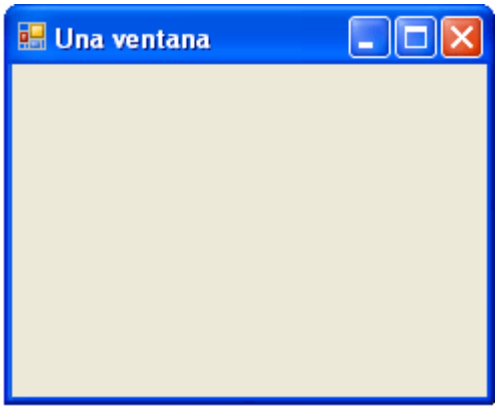

Figura 3.1: Ventana estándar de Windows definida mediante la clase Form de .NET

Ejemplo 3.2 Un programa con ventanas

En este ejemplo vamos a desarrollar un programa que muestra una ventana en pantalla, para de esta manera observar el nivel de abstracción que .NET y C# han puesto a disposición del programador para construir ventanas de una forma muy sencilla. Sin embargo, este solo es una primera aproximación para mostrar como este tipo de aplicaciones en C# utilizan clases y objetos que se manipulan como cualquier otro que ya se haya utilizado en otros ejemplos. La programación de este tipo de programas se describe con mayor detalle en una sección posterior, por ahora solo vamos a realizar un primer intento.

La clase en la cual se encuentra contenida la abstracción de una ventana tipo Windows se llama **Form** y hace parte del espacio de nombres **System.Windows.Forms**. Por lo tanto en el encabezado del archivo fuente se debe incluir este espacio de nombres, así:

```
using System.Windows.Forms;
```

Como se vio anteriormente, para definir un objeto se hace un llamado al nombre de la clase seguido del nombre de la variable que identifica al objeto, en este caso la variable se identificará como *ventana*.

```
Form ventana.
```

Seguidamente se asigna la memoria necesaria para la colocación del objeto.

```
ventana = new Form();
```

Para colocar un título a la ventana el objeto posee la propiedad ***Text*** que permite asignar una cadena de texto que será mostrada en la barra de títulos.

```
ventana.Text = "Hola ventana...";
```

Por último, los objetos del tipo **Form** necesitan informar al sistema en que momento se deben dibujar en pantalla. Para esto cuentan con un método que se encarga de está tarea, el cual se identifica como ***Show***. Para nuestro caso sería,

```
ventana.Show();
```

Con las anteriores especificaciones, estamos listos para construir un programa de muestre una ventana. Este quedaría como sigue:

```
using System.Windows.Forms;

public class HolaVentana
{
      static void Main()
      {
            Form ventana;
            ventana = new Form();
            ventana.Text = "Hola ventana...";
            ventana.Show();
      }
}
```

Pero, en estas condiciones el programa ejecuta todas las líneas de código hasta encontrar el cierre del método *Main* y finaliza. Cuando un programa llega al cierre del método *Main*, como lo hace este programa de consola, finaliza la ejecución y elimina absolutamente todo, incluyendo en este caso la variable *ventana*. Como consecuencia, a pesar de que el método Show dibuja la ventana en pantalla, inmediatamente esta es eliminada y no podremos visualizarla. Para corregir el problema, en esta ocasión y solo para este ejercicio, se introducirá una suspensión de la ejecución mediante el método estático ***Sleep*** de la clase ***Thread***.

```
Thread.Sleep(5000);
```

El parámetro pasado al método ***Sleep***, le indica la cantidad de milisegundos que se debe suspender la ejecución.

La clase ***Thread*** se encuentra definida en el espacio de nombres ***System.Threading***, y su uso en este ejemplo solo es un truco para lograr lo que queremos mostrar con el mismo, ya que su funcionalidad tiene una destinación totalmente diferente que se cubre el la llamada programación multihilo.

En definitiva el programa quedará como sigue:

```
/* Archivo: Ejemplo32.cs  */

using System.Windows.Forms;
using System.Threading;

public class HolaVentana
{
      static void Main()
      {
            Form ventana;
            ventana = new Form();
            ventana.Text = "Hola ventana...";
            ventana.Show();
            Thread.Sleep(5000);
      }
}
```

Compile este programa con la instrucción,

```
csc /out:ventana.exe ejemplo32.cs
```

Al ejecutar nuestro programa, se mostrará en pantalla una ventana dotada con todos los elementos gráficos y funcionales que caracterizan a una ventana estándar de Windows, la cual permanecerá por 5 segundos y luego finalizará, devolviendo el control a la consola.

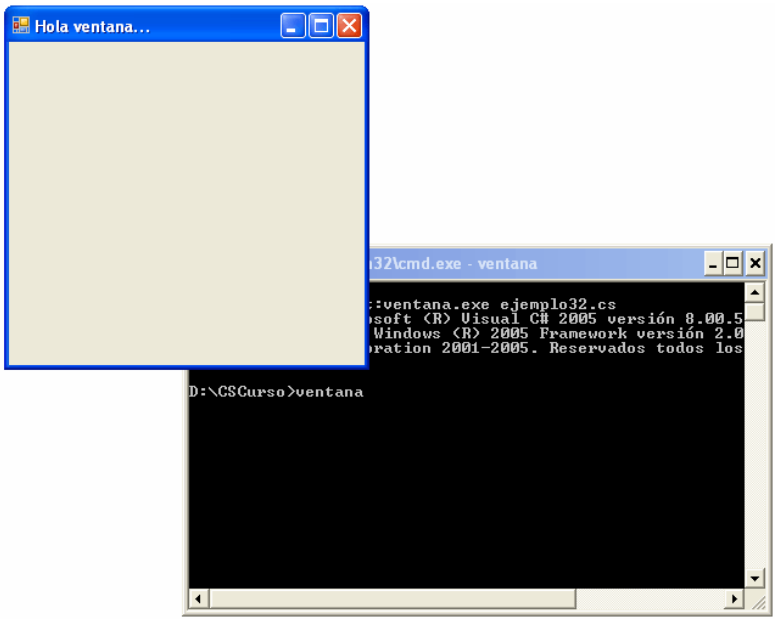

Figura 3.2: Ventana mostrada por un programa de consola.

El desarrollo de este programa nos ha dejado ver como la abstracción realizada por la clase ***Form*** le permite al programador despreocuparse de los detalles técnicos inherentes al dibujado de ventanas en pantalla y las funcionalidades que poseen cada uno de sus componentes, y centrarse mejor en detalles de diseño, que a la larga son más importantes para la finalidad del aplicativo.

De todas maneras la forma como hemos programado esta aplicación no es la utilizada en las aplicaciones estándar que se basan en ventanas ya que la suspensión de la ejecución, generada por la clase *Thread*, congela toda la funcionalidad de la ventana.

Encapsulamiento

El *encapsulamiento* hace referencia a la protección que debe hacer una clase de los elementos que la conforman, especialmente de aquellos que contienen los datos. En general una clase es una *capsula blindada* que preteje a sus miembros del mundo exterior y solo permite acceder a ellos mediante mecanismos que garanticen su adecuada modificación o lectura. Establecer esos mecanismos es una tarea del programador.

Desde la perspectiva de la encapsulación, toda clase debe estar constituida por dos partes básicas: una interfaz y una implementación. La *interfaz* son los elementos que permitirán a los objetos entrar en contacto con el exterior, y la *implementación* son los mecanismos que permiten establecer el comportamiento deseado en cada uno de los objetos que se definan a partir de dicha clase.

Para hacer las cosas más prácticas vamos a considerar una clase llamada *Estudiante*, que nos permitirá manipular los datos de un grupo de estudiantes en una base de datos. Esta clase bien podría definirse así:

```
class Estudiante
{
      public string codigo;
      public string nombres;
}
```

Si se va a utilizar esta clase para procesar datos que serán almacenados en una base de datos, no podemos perder de vista que en este tipo de trabajo existen algunas restricciones que no pueden desconocerse a la hora de leer o registrar cada uno de ellos. Supongamos que se ha establecido el siguiente diccionario de datos:

NOMBRE DEL DATO	TIPO	DESCRIPCIÓN
Código	Cadena de texto	Cadena formada por cuatro dígitos
Nombres	Cadena de texto	Cadena formada por un máximo de 50 caracteres que contiene los nombres y apellidos del estudiante

Al definir en la clase Estudiante como públicos los campos `codigo` y `nombres`, se han dejado totalmente desprotegidos y accesibles al mundo exterior. Así las cosas, la clase Complejo no es para nada una cápsula protectora que brinde seguridad a sus campos, con los posibles errores que esto pueda ocasionar.

Supongamos que a partir de esta clase se define un objeto de *Estudiante* llamado `alumno`, así:

```
Estudiante alumno = new Estudiante();
```

Alguien que haga uso de esta clase podría intentar asignar el código del estudiante mediante una instrucción similar a la siguiente:

```
alumno.codigo = "1";
```

Tal como está definida la clase Estudiante, este valor ingresado es un código perfectamente válido, pero de acuerdo a las condiciones establecidas en el diccionario de datos, que establecen que el código debe ser una cadena de caracteres de cuatro dígitos, este valor debería ser rechazado. Obviamente, un programador, que sabe lo que

está haciendo, difícilmente va a cometer este error, pero si esta clase se va a utilizar para leer datos tecleados por un usuario, las cosas pueden ser a otro precio.

Supongamos que la variable `alumno` se utiliza en una instrucción como la siguiente:

```
alumno.codigo = Console.ReadLine();
```

En esta situación se corre el riesgo que el usuario ingrese cualquier cadena de texto como código del estudiante y el sistema la acepte sin mayor problema. Una posible solución que puede adoptar un programador sería realizar una validación como la siguiente:

```
alumno.codigo = Console.ReadLine();
// Revisar si la longitud del código es válida
if (alumno.codigo.Length != 4)
      MessageBox.Show("El código ingresado no es válido");
```

En este caso se determina si la longitud del código es diferente de 4, y en caso afirmativo se muestra un mensaje en pantalla informando su invalidez. Adicional a esto habría que verificar que la cadena este conformada únicamente por dígitos. De esta forma el programador que utilice la clase Estudiante estaría obligado a realizar esta y otras validaciones cada vez que utilice una variable de este tipo.

En aplicaciones reales, la clase Estudiante puede ser utilizada muchas veces para definir variables que se utilicen en la recolección de datos. Obligar al programador a realizar este tipo de validaciones cada vez que necesite utilizarla sería un atentado en contra del encapsulamiento y la misma abstracción ya que la clase no es capaz de resolver sus propios problemas y deja que otras instancias sean quienes lo hagan.

Un buen encapsulamiento debe garantizar que este tipo de problemas no preocupen la atención de quien vaya a usar esta clase. Para ello es necesario establecer un mecanismo que antes de asignar un valor a un campo, permita validarlo. Y una clase es precisamente una capsula que se encarga en forma interna de verificar el correcto funcionamiento de cada uno de los elementos de la abstracción que representa.

En general, el encapsulamiento exige que todos los campos de una clase se establezcan como privados, y la lectura o modificación de sus valores se realice a través de funciones, que de ser necesario realicen en el momento justo las verificaciones del caso para asegurar que no se han producido infracciones a la abstracción que representa.

El mecanismo de acceso a los datos, que pone a disposición del programador el lenguaje C#, son las propiedades. Una *propiedad* no es más que una función que tiene comunicación directa con uno o varios campos y se encarga de asignar, modificar o leer los valores contenidos en ellos.

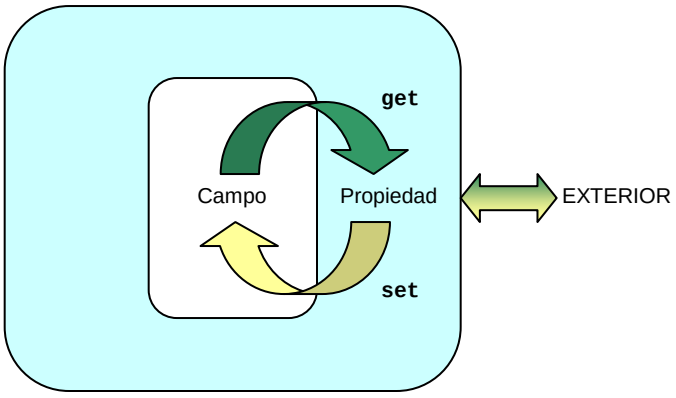

Figura 3.3: Una *propiedad* sirve de vínculo entre el exterior y los campos de un objeto

Una propiedad tiene un aspecto semejante a un método. Se encarga de leer un valor contenido en un campo (**get**) y mostrarlo hacia el exterior; o al contrario, leer un valor desde el exterior y asignarlo a un campo (**set**). En general su sintaxis es como sigue:

```
public tipo NombrePropiedad
{
      get
      {
            // Devuelve con return el valor de un campo
      }
      set
      {
            // Asigna el valor de value al campo
      }
}
```

El descriptor **get** devuelve al exterior el valor contenido en el campo que manipula la propiedad. Este descriptor es equivalente a leer el valor de un campo. Utiliza la palabra clave **return** para identificar el campo devuelto.

El descriptor **set** se encarga de asignar a un campo un valor proveniente desde el exterior. Para ello utiliza el parámetro implícito **value** que contiene el valor que se haya asignado a la propiedad desde el exterior.

Ahora ya contamos con las herramientas necesarias para encapsular los campos de nuestra clase Estudiante. Para empezar cambiamos el nivel de accesibilidad de cada uno de ellos a privado. Así:

```
class Estudiante
{
      // Campos
      private string codigo;
      private string nombres;
}
```

Y en seguida definimos dos propiedades que sirven de medio para comunicarse con cada uno de ellos. Llamaremos a estas propiedades, `Codigo` y `Nombres`, respectivamente. Es una buena práctica llamar a las propiedades con el mismo nombre de los campos que van a manipular.

```
class Estudiante
{
      // Campos
      private string codigo;
      private string nombres;

      // Propiedades
      public string Codigo
      {
            get
            {
                  return codigo;
            }
            set
            {
                  codigo = value;
            }
      }
```

```
        public string Nombres
        {
            get
            {
                return nombre;
            }
            set
            {
                nombre = value;
            }
        }
    }
```

Una buena técnica de programación es llamar a las propiedades por un nombre semejante al del campo que representan. En este caso, teniendo en cuenta el hecho que C# es sensible a las mayúsculas y minúsculas, hemos aprovechado esta característica para dar un nombre similar tanto al campo como a su propiedad asociada, diferenciándolos únicamente en la letra inicial. Esto nos permitirá desarrollar un código más legible y fácil de entender.

Un detalle importante, es tener en cuenta que tanto el campo como la propiedad que lo representa debe ser del mismo tipo. En este caso el campo `codigo` es de tipo **string** y la propiedad `Codigo` también lo es. Lo mismo sucede con el campo `nombres` y la propiedad `Nombres`.

Con el uso de las propiedades hemos encapsulado los campos de la clase Estudiante, aislándolos del mundo exterior, pero aún falta mejorar su encapsulamiento y su nivel de abstracción, dotando a la clase de la capacidad para validar las entradas de datos, y aislar al programador, usuario de esta clase, de los detalles que definen su comportamiento. Esto es lo que vamos a desarrollar en el ejemplo que viene a continuación.

Ejemplo 3.3 La clase estudiante

En este ejemplo vamos a perfeccionar la clase Estudiante para registrar estudiantes en un sistema de información. Lo primero que se debe hacer es verificar que el código sea de cuatro caracteres y luego, que todos los caracteres sean numéricos, específicamente dígitos.

Para realizar la validación vamos a implementar un método booleano que se va a encargar de tal trabajo. Este método recibe una cadena de texto y mediante la línea

```
if (cadena.Length != 4) return false;
```

verifica si su longitud es diferente de 4. En caso afirmativo se retorna el valor **`false`**. La propiedad **Length** establece la longitud de la cadena de texto contenida en la variable.

Para determinar si la cadena contiene algún carácter que no sea dígito es necesario revisarlos uno a uno. Esta tarea la realiza el método booleano **IsDigit** de la clase **Char**, el cual contiene una sobrecarga que recibe dos argumentos: la cadena de texto y el índice que determina la posición del carácter que se va a analizar. El siguiente ciclo realiza un recorrido por cada elemento de la cadena:

```
while(i <= cadena.Length)
{
    i++;
    if (!Char.IsDigit(cadena, i)) return false;
}
```

En caso de encontrarse con un carácter que no es digito se retorna un valor **false** y se da por terminado el ciclo.

Con estos recursos de codificación formamos el método `CodigoValido`, el cual será de accesibilidad privada. De esta manera el mecanismo de validación del código se constituye en un mecanismo interno que queda totalmente aislado del mundo exterior.

```
private bool CodigoValido(string cadena)
{
      int i = 0;

      // Si la longitud del código es diferente de 4...
      if (cadena.Length != 4) return false;

      // Determinar si el código es una cadena de dígitos
      while(i <= cadena.Length)
      {
            i++;
            if (!Char.IsDigit(cadena, i)) return false;
      }

      return true;
}
```

Téngase en cuenta que cuando se ejecuta una instrucción **return**, esta corta la ejecución del método donde se encuentre y devuelve el valor que la sigue. En este caso si el código no esta formado por cuatro caracteres o alguno de ellos no es un dígito se da por terminada la ejecución del método. Por lo tanto solo se llegará a la última línea de código siempre y cuando no se haya ejecutado ninguno de los dos **return false**. Esto garantiza que el código del estudiante únicamente se acepte si pasó todas las pruebas anteriores.

```
/* Archivo: Estudiante.cs */

using System;
using System.Windows.Forms;

public class Estudiante
{
      // Campos
      private string codigo;
      private string nombres;

      // Propiedades
      public string Codigo
      {
            get { return codigo; }
            set
            {
                  if (CodigoValido(value))
                        codigo = value;
                  else
                  {
                        MessageBox.Show("El código " + value + " no es válido.");
                        codigo ="";
                  }
            }
      }
```

```
      public string Nombres
      {
            get { return nombre; }
            set { nombre = value; }
      }

      // Método interno para validar el código
      private bool CodigoValido(string cadena)
      {
            int i = 0;

            // Si la longitud del código es diferente de 4...
            if (cadena.Length != 4) return false;

            // Determinar si el código es una cadena de dígitos
            while(i < cadena.Length)
            {
                  if (!Char.IsDigit(cadena, i)) return false;
                  i++;
            }

            return true;
      }
}
```

Observe que se ha incluido el espacio de nombres **System.Windows.Forms** para generar un mensaje en una caja de mensajes al estilo de Windows. Esta caja de mensajes se encuentra encapsulada en la clase **MessageBox** y se activa mediante el método **Show**.

Realice la compilación de la clase Estudiante en un ensamblado contenido en una librería dinámica, *Estudiante.dll*, mediante la siguiente instrucción de consola:

```
> csc /t:library estudiante.cs
```

Para usar la clase Estudiante, vamos a utilizar el siguiente programa:

```
/* Archivo: Ejemplo33.cs */

using System;

public class DatosEstudiante
{
      static void Main()
      {
            Estudiante alumno = new Estudiante();
            // Error: código de tres caracteres
            alumno.Codigo = "001";
            alumno.Nombre = "Juan";
            // Error: código de 5 caracteres
            alumno.Codigo = "12345";
            alumno.Nombre = "María";
            // Ingreso de datos desde la consola
            alumno.Codigo = Console.ReadLine();
            alumno.Nombre = Console.ReadLine();
      }
}
```

Compile este programa mediante la instrucción,

```
> csc /r:estudiante.dll ejercicio33.cs
```

Cuando el programa es ejecutado, muestra en pantalla un mensaje indicando que el código 001 no es válido, y vuelve a repetir el mensaje para 12345. Si se ingresa por la consola un código válido, no pasa nada y continúa la ejecución. No es aconsejable escribir programas que informen sobre un dato aceptado, por que se supone que esta es la opción obvia.

En general todas las clases deben encapsular sus datos, permitiendo el acceso a los mismos únicamente por medio de propiedades, sin importar si deben o no realizar validaciones de las entradas o las salidas.

Ejemplo 3.4 Las propiedades de la clase Complejo

Sabemos que los elementos básicos de los números complejos son su parte real y su parte imaginaria, pero existe otros elementos que son de gran importancia a la hora de realizar los cálculos matemáticos con ellos, su módulo y su argumento. En el siguiente ejemplo vamos a mejorar la clase Complejo, definida al inicio de este capítulo, encapsulando sus campos y definiendo dos nuevas propiedades para el *módulo* y el *argumento* de este tipo de números.

Inicialmente la clase complejo fue definida en la siguiente forma:

```
class Complejo
{
      // Campos
      public double real;
      public double imaginario;
}
```

Aunque la clase así definida es perfectamente útil para trabajar con números complejos, su diseño desde la metodología orientada a objetos infringe la propiedad del encapsulamiento que debe garantizar cualquier clase. El encapsulamiento debe asegurar que los campos no serán accesibles en forma directa desde el exterior, sino mediante funciones especializadas, que en el caso de C# se conocen como propiedades.

Para encapsular los campos, primeramente debe cambiarse su nivel de accesibilidad a privado, es decir que solo los métodos definidos dentro de la clase podrán acceder a ellos, y luego definirse propiedades que permitan manipular sus valores. De esta manera se crea un envoltorio o encapsulamiento de protección de los campos.

La clase Complejo queda así:

```
class Complejo
{
      // Campos
      private double real;
      private double imaginario;
}
```

Ahora agregamos dos propiedades, `Real` e `Imaginario`, que conforme a su nombre se encargan de establecer comunicación entre los respectivos campos y el mundo exterior:

```
public class Complejo
{
      // Campos
      private double real;
```

```
        private double imaginario;

        // Propiedades
        public double Real
        {
              get { return real; }
              set { real = value; }
        }
        public double Imaginario
        {
              get { return imaginario; }
              set { imaginario = value; }
        }
}
```

Lo anterior no significa que estas sean las únicas propiedades que pueden hacer referencia a los campos de la clase. Por lo general, las propiedades que llevan el nombre de los campos se encargan exclusivamente de leer o modificar su valor, pero pueden definirse otras propiedades que procesen dichos valores y muestren resultados hacia el exterior, sin necesidad que para ellas exista un campo explícitamente definido en la clase. Vamos a definir las propiedades **Módulo** y **Argumento**, que utilizan los dos campos ya definidos, y muestran al exterior resultados relacionados con los números complejos.

El módulo de un complejo se define como el tamaño del vector que corresponde a dicho numero. En otras palabras, el módulo de un número complejo, es la distancia que existe entre el punto que representa dicho numero y el centro de coordenadas. Este puede calcularse mediante el teorema de Pitágoras como,

$$c = \sqrt{a^2 + b^2}$$

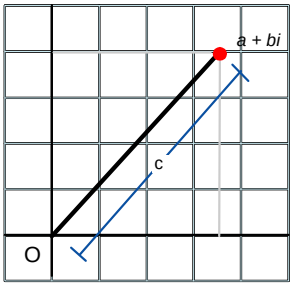

Figura 3.4: El *módulo* de un complejo se interpreta como la distancia entre el complejo y el origen de coordenadas.

En nuestro caso, *a* es la parte real y *b* corresponde a la parte imaginaria del complejo. Por lo tanto, en términos de C# vamos a codificar, el módulo de un complejo, mediante,

```
c = Math.Sqrt(real * real + imaginario * imaginario);
```

Aunque bien podría incluirse el código completo para calcular el módulo dentro de una propiedad, vamos a definir un método interno que se encargue de realizar el respectivo cálculo. Ese método se llamará `Tamano`, así:

```
private double Tamano()
{
      double c;
      c = Math.Sqrt(real*real + imaginario*imaginario);
      return c;
}
```

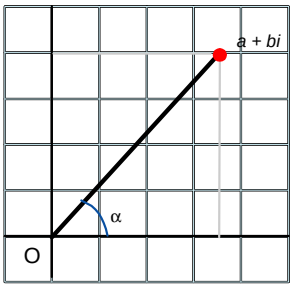

Figura 3.5: El *argumento* de un complejo es el ángulo formado por el vector que representa el complejo y la parte positiva del eje de las abscisas.

La única razón para proceder de esta manera es que en otro cálculo, podría usarse nuevamente el módulo del número complejo, y definido de esta manera puede reutilizarse sin mayor problema.

El argumento de un número complejo hace referencia al ángulo que forma el vector correspondiente a dicho número y la parte positiva del eje horizontal del plano cartesiano, o eje de las abscisas. Teniendo en cuenta la trigonometría, este ángulo puede calcularse mediante la aplicación de la función *arco tangente* a la división de la parte imaginaria entre la parte real, así

$$\alpha = \arctan\left(\frac{b}{a}\right)$$

En C# esta fórmula se puede implementar como,

```
alfa = Math.Atan(imaginario / real);
```

Sin embargo, se debe tener en cuenta que el método **Math.Atan** solo devuelve ángulos comprendidos entre $-\pi/2$ y $\pi/2$. Esto, aunque matemáticamente es válido, puede perjudicar la comprensión intuitiva de algunos cálculos con complejos. Por ejemplo, cuando el número se ubique en el segundo cuadrante, se tendrá un ángulo mayor a $\pi/2$, pero el método nos devolverá un valor negativo comprendido entre $-\pi/2$ y 0, que corresponde al ángulo β de la figura 3.6. Es decir que,

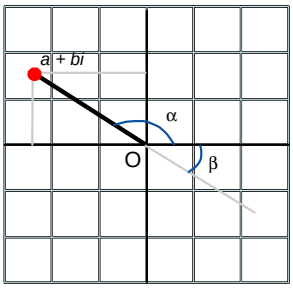

Figura 3.6: Número complejo en el segundo cuadrante.

$$\beta = \arctan\left(\frac{b}{a}\right)$$

Pero el ángulo de nuestro interés sigue siendo α y la implementación que se requiere es para este ángulo. Según la gráfica de la figura 3.6, la suma de los valores absolutos de los ángulos α y β es igual a π radianes (o 180º). Teniendo en cuenta que en el segundo cuadrante, el valor devuelto para β es un valor negativo, podemos escribir esta propiedad como,

$$\alpha - \beta = \pi$$

con lo cual se obtiene,

$$\alpha = \pi + \beta$$

$$\alpha = \pi + \arctan\left(\frac{b}{a}\right)$$

Por lo tanto, la implementación en términos de C# queda como,

```
alfa = Math.PI + Math.Atan(imaginario / real);
```

Con un análisis similar se puede determinar la implementación para calcular el argumento de un complejo ubicado en el tercer cuadrante del plano cartesiano. El resultado obtenido debe ser el siguiente:

```
alfa = - Math.PI + Math.Atan(imaginario / real);
```

Con todo este análisis ya podemos implementar un método interno para calcular el ángulo de un complejo. A este método le llamaremos `Angulo`, así:

```
private double Angulo()
{
      double alfa;

      if (real > 0)
            alfa = Math.Atan(imaginario / real);
      else if (real < 0)
            if (imaginario > 0)
                  alfa = Math.PI + Math.Atan(imaginario/real);
            else
                  alfa = -Math.PI + Math.Atan(imaginario/real);
```

```
            else
                  if (imaginario > 0)
                        alfa = Math.PI / 2;
                  else
                        alfa = - Math.PI / 2;

            return alfa;
      }
```

Finalmente, se definen dos propiedades que se encargarán de mostrar al mundo exterior los valores devueltos por los dos métodos que acabamos de crear. Aquí las propiedades no actuaran directamente sobre campos representados por variables sino valores devueltos por métodos internos de la clase, otra razón por la cual existen las propiedades dentro del mundo de la programación orientada a objetos con C#. Dentro de la pretensión de lograr un alto nivel de abstracción también es importante seleccionar un nombre significativo a las propiedades, en este caso, como es obvio, se llamarán **Modulo** y **Argumento**, respectivamente.

Tanto la propiedad Módulo, como la propiedad Argumento, únicamente se encargarán de ejecutar su respectivo método y devolver un valor, por lo tanto no deben admitir ingresar un dato hacia el interior. En este caso las dos propiedades solo deben manejar el descriptor **get**, que automáticamente las convierte en propiedades de solo lectura.

```
public double Modulo
{
      get { return Tamano(); }
}

public double Argumento
{
      get { return Angulo(); }
}
```

El archivo fuente de la clase *Complejo* por ahora queda como sigue:

```
/* Archivo: Complejo.cs */

using System;
public class Complejo
{
      // Campos
      private double real;
      private double imaginario;

      // Propiedades
      public double Real
      {
            get { return real; }
            set { real = value; }
      }

      public double Imaginario
      {
            get { return imaginario; }
            set { imaginario = value; }
      }

      public double Modulo
      {
            get { return Tamano();}
      }
```

```
      public double Argumento
      {
            get { return Angulo(); }
      }

      // Métodos privados
      private double Tamano()
      {
            double c;
            c = Math.Sqrt(real * real + imaginario * imaginario);
            return c;
      }

      private double Angulo()
      {
            double alfa;
            if (real > 0)
                  alfa = Math.Atan(imaginario / real);
            else if (real < 0)
                  if (imaginario > 0)
                        alfa = Math.PI + Math.Atan(imaginario / real);
                  else
                        alfa = - Math.PI + Math.Atan(imaginario / real);
                  else
      if (imaginario > 0)
                        alfa = Math.PI / 2;
                  else if (imaginario < 0)
                        alfa = - Math.PI / 2;
                  else
                        alfa = 0;
            return alfa;
      }
}
```

Guarde este archivo con el nombre Complejo.cs y compílelo como librería dinámica mediante el comando,

```
> csc /t:library Complejo.cs
```

Aunque es posible que al lector no le interese mucho el análisis de tipo matemático realizado para este ejemplo, si es importante tener en cuenta que en un proyecto real el análisis que se haga del mundo que se va a modelar mediante una clase, es definitivo para lograr una buena abstracción que permita desarrollar componentes de software eficientes y altamente reutilizables.

Con el ensamblado que hemos desarrollado ya podemos manipular números complejos y realizar algunos cálculos con ellos. El siguiente programa se encarga de leer desde el teclado los valores de las partes real e imaginaria de un complejo y mostrar en pantalla su módulo y argumento.

```
/* Archivo: Ejemplo34.cs  */

using System;

public class NumeroComplejo
{
      public static void Main()
      {
            Complejo z = new Complejo();
```

```
            Console.Write("a = ");
            z.Real = Convert.ToDouble(Console.ReadLine());
            Console.Write("b = ");
            z.Imaginario = Convert.ToDouble(Console.ReadLine());

            Console.WriteLine("Módulo = {0}", z.Modulo);
            Console.WriteLine("Argumento = {0} radianes", z.Argumento);

            Console.Write("Presione cualquier tecla para continuar . . . ");
            Console.ReadKey(true);
        }
}
```

Compile este programa con el comando,

```
> csc /r:complejo.dll ejercicio34.cs
```

Hemos realizado un largo recorrido por la teoría de la programación orientada a objetos, e incluso los mismos conceptos matemáticos sobre los números complejos, para lograr un nivel aceptable de abstracción de la plantilla que nos permita construir y manipular este tipo de números dentro del lenguaje C# en forma correcta y consistente. Sin embargo, por ahora las variables de tipo Complejo solo sirven para almacenar la parte real e imaginaria de un número complejo pero les hace falta lo esencial de un elemento matemático: la capacidad de responder adecuadamente ante las operaciones matemáticas. Este y otros aspectos serán implementados en los ejemplos que acompañan a las secciones que vienen a continuación.

Modularidad

La *modularidad* es la propiedad, de la programación orientada a objetos, que permite dividir una aplicación de software en componentes más pequeños, generalmente llamados módulos. Cada uno de los componentes debe ser lo más independiente posible tanto de la aplicación como de los demás componentes.

Este aspecto de la programación orientada a objetos es el que nos obliga a dedicar más tiempo al diseño y ser muy cuidadosos en la forma como se debe asumir el desarrollo de una aplicación de software. Aunque existe una amplia documentación sobre el tema en libros, revistas, Internet o en las mismas cátedras universitarias, solo la experiencia será quién al final nos enseñe la forma como debemos subdividir una aplicación en diferentes componentes de software, de tal manera que podamos sacar el mayor provecho posible de los mismos. Una estrategia práctica que puede ser útil en la escogencia de una buena modularidad es no pensar que el proyecto en el cual se está trabajando actualmente es la única aplicación que se va a desarrollar, sino que existirán muchas otras en el futuro con características diferentes o semejantes, y determinar que partes de la actual me pueden ser útiles en esas aplicaciones de software. Un buen programador, a medida que desarrolla programas de computador, debe ir construyendo sus propias bibliotecas de componentes de software que le puedan ser útiles en otras aplicaciones o a otros programadores, sin necesidad de tener que recurrir a modificaciones de su programación.

El lenguaje C# y .NET han puesto a disposición del programador una gran cantidad de componentes y la posibilidad de crear los suyos agrupándolos en los denominados *espacios de nombres*, que impiden la posibilidad de crear conflictos por la existencia de nombres duplicados de las clases. En la práctica toda aplicación debe organizarse en uno o varios espacios de nombres. Los componentes que se consideren más genéricos pueden hacer parte de un espacio de nombres diferentes al de la aplicación para de esta manera facilitar su portabilidad a otros proyectos.

Un aspecto importante, consecuencia de una buena modularidad es la posibilidad de realizar, de una manera rápida y eficaz, el mantenimiento de una aplicación. Muchas veces los clientes y los programadores de un proyecto de software identifican deficiencias en la ejecución de la aplicación, incluso mucho tiempo después de haber sido puesta en funcionamiento. Si la aplicación está dividida en componentes, bastará con identificar el componente que falló, corregirlo, compilarlo y reemplazarlo en la aplicación donde se esté utilizando. En el caso de C# y .NET esta tarea se facilita enormemente, ya que solo bastará con reemplazar el ensamblado o ensamblados que hagan parte del componente utilizando copiar y pegar.

Ejemplo 3.5 Un componente para validar códigos

En el siguiente ejemplo se va a desarrollar un componente que se encarga de generar y validar el código para un registro de datos.

En el registro de datos, en una base de datos, se suele utilizar la codificación de cada uno de los registros que ingresan al sistema, para facilitar su posterior búsqueda y modificación. En la construcción de dichos códigos que se acostumbran a utilizar una serie de reglas que les permite a los administradores del sistema almacenar información relevante para las consultas y mantenimiento del sistema. Una forma de crear los códigos es mediante la utilización de una cadena de caracteres conformada por un prefijo y un sufijo. El prefijo, por lo general, contiene información que identifica a un grupo de registros como de la misma familia, mientras que el sufijo se corresponde con el orden de ingreso al sistema. Así, por ejemplo, en una universidad los estudiantes de una carrera bien podrían codificarse con un prefijo que corresponde al código de la facultad o programa al cual pertenecen, mientras que el sufijo bien podría corresponder al orden de registro de ingreso a la universidad. Adicional a lo anterior es conveniente, por cuestiones de organización, dar una longitud fija a los códigos que se vayan a utilizar.

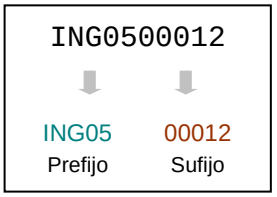

Figura 3.7: Ejemplo de código compuesto por un prefijo y un sufijo.

El componente de software debe recibir el prefijo, el sufijo y la longitud que tendrá el código, para con base en ellos generar una cadena de texto que se utilizará para crear la cadena de codificación. La cadena de texto generada corresponderá al valor del código.

Inicialmente la clase básica del componente se ve como sigue:

```
public class Codigo
{
      string prefijo = "";
      string sufijo = "";
      int longitud;
      string valor;

      public string Prefijo
      {
            get { return prefijo; }
            set { prefijo = value; }
      }

      public string Sufijo
      {
            get { return sufijo; }
            set { sufijo = value; }
      }

      public int Longitud
      {
            get { return longitud; }
            set { longitud = value; }
```

```
        }

        public string Valor
        {
            get { return valor; }
        }
    }
```

El componente se encargará de verificar que el código generado cumpla con las reglas básicas: debe iniciar por el prefijo establecido, debe terminar con el sufijo y tener una longitud igual a la establecida. En caso que la concatenación del prefijo con el sufijo generen una cadena de longitud inferior a la establecida se debe rellenar con ceros de por medio. En caso que la concatenación genere una cadena de longitud mayor a la establecida se generará un error.

El siguiente método se encarga de validar el cumplimiento de las anteriores condiciones:

```
    public bool Valido()
    {
        int longitudPrefijo = prefijo.Length;
        int longitudSufijo = sufijo.Length;
        string cadena = sufijo;
        int i = longitudSufijo;
        if (longitudPrefijo + longitudSufijo == longitud)
            valor = prefijo + sufijo;
        else if(longitudPrefijo + longitudSufijo < longitud)
        {
            // Se rellena con ceros de por medio
            while (i < longitud - longitudPrefijo)
            {
                cadena = "0" + cadena;
                i++;
            };
            valor = prefijo + cadena;
        }
        else
        {
            MessageBox.Show("No se pudo generar...");
            return false;
        }
        MessageBox.Show("El código es: " + valor);
        return true;
    }
```

Se ha definido un método público para permitirle al usuario del componente verificar la validez del código que se vaya a generar en el momento justo. Además, se ha se lo ha definido de tipo booleano para facilitar una toma de decisiones en el momento de validarlo. Si el código es válido se ejecuta alguna acción con el registro, en caso contrario se suspende.

Integremos las anteriores partes en el archivo fuente *Ejemplo35.Codigo.cs*:

```
/* Archivo: Ejemplo35.Codigo.cs*/

using System;
using System.Windows.Forms;

namespace Ejemplo35
{
    public class Codigo
```

```
	{
		string prefijo = "";
		string sufijo = "";
		int longitud;
		string valor;

		public string Prefijo
		{
			get { return prefijo; }
			set { prefijo = value; }
		}

		public string Sufijo
		{
			get { return sufijo; }
			set { sufijo = value; }
		}
		public int Longitud
		{
			get { return longitud; }
			set { longitud = value; }
		}
		public string Valor
		{
			get
			{
				return valor;
			}
		}
		public bool Valido()
		{
			int longitudPrefijo = prefijo.Length;
			int longitudSufijo = sufijo.Length;
			string cadena = sufijo;
			int i = longitudSufijo;

			if (longitudPrefijo + longitudSufijo == longitud)
				valor = prefijo + sufijo;
			else if (longitudPrefijo + longitudSufijo < longitud)
			{
				// Se rellena con ceros de por medio
				while (i < longitud - longitudPrefijo)
				{
					cadena = "0" + cadena;
					i++;
				};
				valor = prefijo + cadena;
			}
			else
			{
				MessageBox.Show("El código no se pudo generar...");
				return false;
			}

			MessageBox.Show("El código es: " + valor);
			return true;
		}
	}
}
```

Compile este archivo en una librería dinámica mediante la instrucción:

```
> csc /t:library Ejemplo35.Codigo.cs
```

Y ahora nuestro programa para hacer uso del ensamblado que acabamos de crear:

```
using System;

namespace Ejemplo35
{
      public class Programa
      {
            public static void Main(string[] args)
            {
                  Codigo c = new Codigo();
                  string codigo;
                  c.Prefijo = "ING05";
                  c.Sufijo = "12";
                  c.Longitud = 15;
                  if (c.Valido()) codigo = c.Valor;
                  Console.WriteLine("Código = " + c);
                  Console.Write("Presione una tecla...");
                  Console.ReadKey(true);
            }
      }
}
```

Para compilar este programa se debe aplicar una instrucción como la siguiente:

```
> csc /r:Ejemplo35.Codigo.dll ejemplo35.cs
```

Al ejecutar el programa, *Ejemplo35.exe*, se muestra en pantalla una caja de mensajes informando sobre la validez del código generado. Observe que, este mensaje es generado directamente por el componente generador de códigos. Como muestra de la capacidad de abstracción y encapsulamiento, esta es una muy buena opción, ya que se cuenta con una plantilla que produce objetos altamente autónomos. Sin embargo, en este punto el programador debe tomar una decisión y definir en que entorno puede llegar a ser utilizado su componente de software.

Tal como se ha programado el ensamblado *Ejemplo35.Codigo.dll* solo podrá ser utilizado en aplicaciones de interfaces gráficas tipo Windows. La caja de mensajes es generada por la clase **MessageBox** que encapsula las funciones del sistema gráfico del sistema operativo Windows. Por lo tanto, la reutilización del componente se limita a aplicaciones que se ejecuten únicamente en los sistemas operativos de la familia Windows.

Si el interés del programador es contar con un componente que pueda ser utilizado en aplicaciones que se vayan a ejecutar en cualquier sistema operativo que ejecute aplicaciones C#, lo primero que deberá hacer es revisar la modularidad de sus componentes, específicamente en cuanto a dependencias de otros componentes se refiere. Incluso es posible que se desee contar con un componente que pueda ser utilizado en aplicaciones web.

En la vida práctica, es el programador quién debe tomar la mejor alternativa: deja las cosas como están, y dispone de un componente validador para programas tipo Windows, con un alto nivel de abstracción y encapsulamiento, o crea un componente más genérico que le sea útil en diversos tipos de aplicaciones. Solo la práctica y el objetivo que se busque en un proyecto de software serán quienes dicten las reglas que se deben seguir en lo que respecta a la forma como asumamos la modularidad.

Ejemplo 3.6 Validador universal de códigos

En este ejemplo vamos a crear el mismo componente del ejemplo anterior, pero haciéndolo más genérico. En otras palabras, vamos a modificar el archivo fuente de tal manera que no dependa del espacio de nombres **System.Windows.Forms** y los cambios que esto implica en la clase. El nuevo componente se encargará de devolver un mensaje de texto a través de una propiedad que llamaremos *Estado*.

La nueva clase, que definirá el componente validador de códigos, ya no podrá mostrar un mensaje a través de la ventana caja de mensajes. En consecuencia vamos a agregar un nuevo campo que se identificará como *estado* y que será manipulado por la propiedad *Estado*. Esta propiedad debe ser de solo lectura por que no necesita recibir un mensaje sino únicamente mostrarlo hacia el exterior. Su implementación es como sigue:

```
public string Estado
{
      get { return estado; }
}
```

Entonces, la clase que nos permite definir los objetos generadores de códigos queda así:

```
/* Archivo: Ejemplo36.Codigo.cs*/

namespace Ejemplo36
{
      public class Codigo
      {
            string prefijo = "";
            string sufijo = "";
            int longitud;
            string valor;
            string estado;

            // Propiedades
            public string Prefijo
            {
                  get { return prefijo; }
                  set { prefijo = value; }
            }

            public string Sufijo
            {
                  get { return sufijo; }
                  set { sufijo = value; }
            }

            public int Longitud
            {
                  get { return longitud; }
                  set { longitud = value; }
            }

            public string Valor
            {
                  get { return valor; }
            }
```

```
            public string Estado
            {
                  get { return estado; }
            }

            // Método
            public bool Valido()
            {
                  int longitudPrefijo = prefijo.Length;
                  int longitudSufijo = sufijo.Length;
                  string cadena = sufijo;
                  int i = longitudSufijo;

                  if (longitudPrefijo + longitudSufijo == longitud)
                        valor = prefijo + sufijo;
                  else if (longitudPrefijo + longitudSufijo < longitud)
                  {
                        // Se rellena con ceros de por medio
                        while (i < longitud - longitudPrefijo)
                        {
                              cadena = "0" + cadena;
                              i++;
                        };
                        valor = prefijo + cadena;
                  }
                  else
                  {
                        estado = "El código no se pudo generar...";
                        return false;
                  }

                  estado = "El código es: " + valor;
                  return true;
            }
      }
}
```

Compile este archivo en una librería dinámica mediante la instrucción:

```
> csc /t:library Ejemplo36.Codigo.cs
```

El siguiente programa hace uso del componente que acabamos de crear:

```
/* Archivo: Ejemplo36.cs */

using System;
using System.Windows.Forms;
using Ejemplo36;

public class Programa
{
      public static void Main(string[] args)
      {
            codigo c = new codigo();
            string código;
            Console.Write("Prefijo: ");
            c.Prefijo = Console.ReadLine();
            Console.Write("Sufijo: ");
            c.Sufijo = Console.ReadLine();
```

```
            Console.Write("Longitud: ");
            c.Longitud = Convert.ToInt32(Console.ReadLine());
            if (c.Valido())
            {
                  codigo = c.Valor;
                  MessageBox.Show(c.Estado);
            }
            else
                  MessageBox.Show(c.Estado);
      }
}
```

Compile el programa con la instrucción,

```
> csc /r:ejemplo36.codigo.dll ejemplo36.cs
```

El nuevo componente ya no encapsula la posibilidad de mostrar un mensaje por su propia cuenta. Ahora es el programador, que haga uso del componente, quién debe preocuparse por este trabajo, si es que lo necesita. Sin embargo, se ha ganado en modularidad, ya que el nuevo ensamblado puede utilizarse en cualquier tipo de aplicación .NET que funcione bajo el sistema operativo Windows, ya sea de consola, para entorno de ventanas o web.

Herencia

La herencia es la característica más importante de la programación orientada a objetos por que permite crear clases que se derivan de otras clases y este es uno de los aspectos vitales en la *reutilización de componentes*.

Específicamente, la herencia es la capacidad que ofrece un lenguaje de programación de poder crear nuevas clases a partir de clases existentes, aprovechando el código de estas últimas. La herencia supone la existencia de una clase base y una o más clases derivadas. Tomando la analogía del mundo real se dice que una clase derivada hereda las características de la clase base, agregando sus propios elementos.

En la práctica cuando un programador construye una clase, heredando las características de otra ya existente, lo que está haciendo es utilizar la estructura de esa clase e incorporándola a la suya, y adicional le agrega o incluso le quita algunas características que requiera su desarrollo.

En la literatura sobre programación orientada a objetos existen muchos ejemplos didácticos para intentar explicar la herencia, como por ejemplo que las clases Ave, Felino y Mamífero se derivan de la superclase Animal por que todos estos especimenes comparten características, que se pueden abstraer en la clase base. En el desarrollo de software lo que el programador debe hacer es identificar todos aquellos campos que se repitan en un conjunto de clases y con ellos formar una clase que le sirva como base para las demás. Los elementos que se incluyeron en la clase base, ya no será necesario especificarlos en las clases derivadas, trayendo como consecuencia una simplificación sustancial en el proceso de programación.

Para hacer más didáctica la explicación, supongamos que en una entidad educativa se va a sistematizar, entre otros, los datos del personal vinculado a ella. Para empezar, sabemos que existen estudiantes y trabajadores, cuyos datos los vamos a procesar a través de las clases Estudiante y Trabajador, respectivamente.

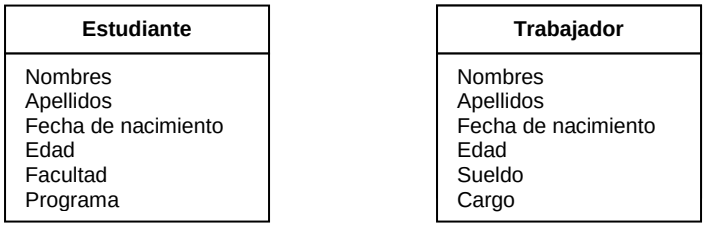

Figura 3.8: Campos de las clases Estudiante y Trabajador

Tanto los estudiantes como los trabajadores tienen elementos o datos comunes, tales como nombres, apellidos, fecha de nacimiento y edad, y otros que no lo son. Aplicar la propiedad de la herencia en este caso, implica rediseñar las clases creando una nueva clase a partir de los campos que se encuentran repetidos en ambas. A esa clase bien podríamos llamarla Persona y su estructura sería como se muestra en la figura 3.9. A partir de esta clase se derivan las clases Estudiante y Trabajador con los campos que les hacen falta a cada una de ellas[1].

El nuevo diseño del conjunto de clases, en este ejemplo, va a simplificar la programación de algunos elementos de las clases. Por ejemplo, la edad, de un estudiante o un trabajador, que se puede calcular con base en la fecha de nacimiento, en este caso bastará con programar su cálculo una sola vez y en la clase base. Los resultados estarán disponibles tanto para los objetos que se declaren a partir de cualquiera de las clases derivadas, como para aquellos que se definan a partir de la misma clase Persona.

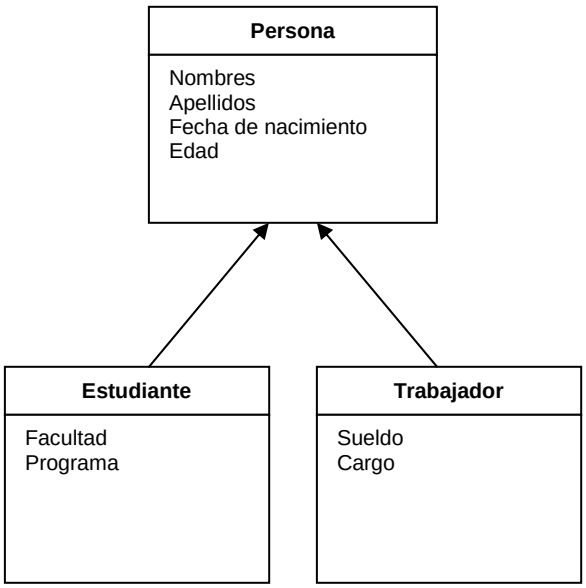

Figura 3.9: Las clases Estudiante y Trabajador se heredan de la clase Persona

La herencia es una propiedad supremamente potente que exige especial cuidado al momento de diseñar un software. Esta propiedad bien utilizada simplifica enormemente la programación y sobre todo permite sacar el máximo provecho a los elementos que pone a disposición, tanto el entorno de desarrollo, como aquellos que sea capaz de crear el ingeniero de software.

Dentro de la programación orientada a objetos existen dos tipos de herencia: simple y múltiple. Existe *herencia simple* cuando una clase solo se deriva de una clase base; y

[1] En los diagramas UML se acostumbra a representar la herencia mediante una flecha que va desde la clase derivada hasta la clase base. La flecha así dibujada establece una relación de tipo *es un*.

herencia múltiple cuando una clase se deriva de dos o más clases base. Este es un aspecto donde C# presenta una debilidad[1], al menos en la opinión de algunos expertos, ya que solo es capaz de soportar la herencia simple. Algunos opinan que un lenguaje que no de soporte a la herencia múltiple pone en duda su pertenencia a la familia orientada a objetos.

Para especificar en C#, que una clase hereda de otra se la define escribiendo su nombre seguido de dos puntos y el nombre de la clase que sirve de base para la herencia. La siguiente es la sintaxis que se debe manejar:

```
class ClaseDerivada : ClaseBase
{
      // Implementación de la clase derivada
}
```

En el ejemplo de nuestro sistema de datos para una entidad educativa, podemos definir la clase base Persona como,

```
class Persona
{
      // Implementación de la clase base
}
```

y las clases Estudiante y Trabajador, que heredan de ella, se debe hacer de la siguiente forma:

```
class Estudiante : Persona
{
      // Elementos propios de esta clase
}

class Trabajador : Persona
{
      // Elementos propios de esta clase
}
```

Ejemplo 3.7 La herencia de las personas

Ya hemos analizado la herencia de la clase Persona con relación a sus derivadas Estudiante y Trabajador. En el siguiente ejemplo vamos a programar un ensamblado con estas clases y luego probaremos su funcionalidad en un pequeño programa que hace uso de ellas.

La mayor parte de la implementación de las clases es simple y, a excepción de la herencia, no tiene nada de diferente a lo que ya se ha realizado en otros ejemplos. Talvez el único detalle diferente, en lo que a programación se refiere, es el registro de la fecha de nacimiento y el cálculo de la edad.

Para registrar la fecha de nacimiento vamos a utilizar un campo de tipo **DateTime**. Este tipo es definido por una estructura que permite establecer tipos de valores que representa fechas y horas comprendidos entre la medianoche (00:00:00) del 1 de enero del año 1 y las 23:59:59 del 31 de diciembre de 9999 d.C. de la Era Cristiana. Por lo tanto, la implementación inicial de la clase Persona queda como sigue:

```
public class Persona
{
```

[1] La limitación de herencia simple la impone .NET ya que solo da soporte a este tipo de herencia. Pero dado que C# es el lenguaje base de .NET, debe soportar todas las criticas de los puristas de la programación orientada a objetos.

```
        // Campos
        string nombres;
        string apellidos;
        DateTime fechaNacimiento;
        int edad;

        // Propiedades
        public string Nombres
        {
              get { return nombres; }
              set { nombres = value; }
        }
        public string Apellidos
        {
              get { return apellidos; }
              set { apellidos = value; }
        }

        public DateTime FechaNacimiento
        {
              get { return fechaNacimiento; }
              set { fechaNacimiento = value; }
        }

        public int Edad
        {
              get { return edad; }
        }
}
```

Para establecer la edad es necesario contar con un método interno que se encargue de calcular la diferencia de tiempo entre la fecha actual y la fecha de nacimiento. Los intervalos de tiempo comprendidos entre dos instantes de tiempo se pueden definir mediante un tipo de valores denominado **TimeSpan**. Adicional, la fecha y hora actuales del sistema se obtiene a través de la propiedad estática **Now** de la estructura **DateTime.** Así las cosas, la edad de una persona puede calcularse mediante el siguiente método:

```
private void CalcularEdad()
{
      TimeSpan intervalo;
      intervalo = DateTime.Now - fechaNacimiento;
      edad = (int)(intervalo.Days / 365.25);
}
```

Un valor del tipo **TimeSpan** almacena un dato en términos de días, horas, minutos, segundos y fracciones de segundo. En este caso solo hemos tomado los días del intervalo de tiempo encontrado. La división por 365.25 hace una conversión a años. Por último el cálculo de la edad debe realizarse después haber leído la fecha de nacimiento, por lo que será llamado por la propiedad *FechaNacimiento*.

La implementación definitiva de la clase Persona queda como sigue:

```
public class Persona
{
      // Campos
      string nombres;
      string apellidos;
      DateTime fechaNacimiento;
      int edad;
```

```
        // Propiedades
        public string Nombres
        {
              get { return nombres; }
              set { nombres = value; }
        }

        public string Apellidos
        {
              get { return apellidos; }
              set { apellidos = value; }
        }

        public DateTime FechaNacimiento
        {
              get { return fechaNacimiento; }
              set {
                    fechaNacimiento = value;
                    CalcularEdad();
              }
        }

        public int Edad
        {
              get { return edad; }
        }

        private void CalcularEdad()
        {
              TimeSpan intervalo;
              intervalo = DateTime.Now - fechaNacimiento;
              edad = (int)(intervalo.Days / 365.25);
        }
  }
```

Las clases Estudiante y Trabajador, que heredan de Persona, solo deben implementar sus campos y propiedades particulares. Un detalle que se debe tener bien en cuenta es el nivel de accesibilidad que deben tener las clases base y sus derivadas. El nivel de accesibilidad de la clase base siempre debe ser mayor o igual al de sus derivadas. Por ejemplo, si se tiene un clase base privada, al pretender derivar una clase de publica se genera un error de compilación.

El siguiente es el archivo definitivo que muestra la implementación de las tres clases de nuestro ejemplo.

```
/* Archivo: Personal.cs */
using System;

// Clase base
public class Persona
{
      // Campos
      string nombres;
      string apellidos;
      DateTime fechaNacimiento;
      int edad;

      // Propiedades
      public string Nombres
      {
```

```
            get { return nombres; }
            set { nombres = value; }
      }

      public string Apellidos
      {
            get { return apellidos; }
            set { apellidos = value; }
      }

      public DateTime FechaNacimiento
      {
            get { return fechaNacimiento; }
            set {
                  fechaNacimiento = value;
                  CalcularEdad();
            }
      }

      public int Edad
      {
            get { return edad; }
      }

      private void CalcularEdad()
      {
            TimeSpan intervalo;
            intervalo = DateTime.Now - fechaNacimiento;
            edad = (int)(intervalo.Days / 365.25);
      }
}

// Clases derivadas
public class Estudiante : Persona
{
      string codigo;
      string facultad;
      string programa;

      public string Codigo
      {
            get { return codigo; }
            set { codigo = value; }
      }

      public string Facultad
      {
            get { return facultad; }
            set { facultad = value; }
      }

      public string Programa
      {
            get { return programa; }
            set { programa = value; }
      }
}

public class Trabajador : Persona
{
      int sueldo;
```

```
      string cargo;

      public int Sueldo
      {
            get { return sueldo; }
            set { sueldo = value; }
      }

      public string Cargo
      {
            get { return cargo; }
            set { cargo = value; }
      }
}
```

Guarde este archivo con el nombre *Personal.cs* y compílelo mediante el comando,

```
> csc /t:library personal.cs
```

El siguiente programa hace uso de las clases Estudiante y Trabajador:

```
/* Archivo: Ejemplo37.cs */

using System;

public class Programa
{
      public static void Main(string[] args)
      {
            Trabajador empleado = new Trabajador();
            Estudiante alumno = new Estudiante();

            // Datos de un estudiante
            Console.WriteLine("Estudiante");
            alumno.Codigo = "01";
            alumno.Nombres = "Juan";
            alumno.Apellidos = "Rodríguez";
            alumno.FechaNacimiento = Convert.ToDateTime("12/05/1990");
            Console.Write("Código   : {0}\n", alumno.Codigo);
            Console.Write("Nombres  : {0}\n", alumno.Nombres);
            Console.Write("Apellidos: {0}\n", alumno.Apellidos);
            Console.Write("Fecha N  : {0}\n", alumno.FechaNacimiento);
            Console.Write("Edad     : {0} años\n", alumno.Edad);

            // Datos de un trabajador
            Console.WriteLine("Trabajador");
            Console.Write("Nombres: ");
            empleado.Nombres = Console.ReadLine();
            Console.Write("Apellidos: ");
            empleado.Apellidos = Console.ReadLine();
            Console.Write("Fecha N: ");
            empleado.FechaNacimiento = Convert.ToDateTime(Console.ReadLine());
            Console.Write("Sueldo: ");
            empleado.Sueldo = Convert.ToInt32(Console.ReadLine());
            Console.Write("Edad: {0}", empleado.Edad);
      }
}
```

Compile este programa con la instrucción,

```
> csc /r:personal.dll ejemplo37.cs
```

Polimorfismo

En general se dice que el *polimorfismo* es la posibilidad de que una entidad tome *muchas formas*. En términos de programación orientada a objetos, y de forma práctica, el polimorfismo permite hacer referencia a objetos de diferentes clases por medio de una misma operación, la cual se ejecuta y aplica de acuerdo al objeto que la invoca.

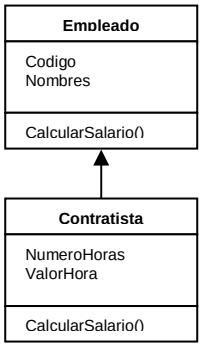

Fig. 3.10: La clase Contratista hereda de la clase Empleado y redefine el método *CalcularSalario()*.

Aunque el concepto de polimorfismo tiene diversos matices dentro de la programación orientada a objetos, básicamente se trabaja relacionado con el concepto de herencia. Si una clase base implementa una serie de miembros (propiedades y métodos), que pueden ser heredados por otras clases, el polimorfismo es la propiedad que le permite al programador, que haga uso de las clases derivadas, modificar el comportamiento de tales miembros ya sea parcial o totalmente. Por ejemplo, en la figura 3.10 tenemos una clase Empleado y el método `CalcularSalario()` que se encarga de calcular el valor salarial asignado a cada empleado de una empresa. Supongamos que los datos de un contratista se procesan en forma similar a los de un empleado corriente de la empresa, pero su asignación salarial se calcula con base en factores que no se tienen en cuenta para el primero, entonces el polimorfismo nos permite implementar una nueva clase a la que hemos llamado *Contratista*, en la cual puede sobrescribirse el método `CalcularSalario()`, acorde al cálculo que se debe realizar para los datos que procesará este tipo. Los objetos que se definan con la clase *Empleado* utilizarán el método `CalcularSalario()` de esa clase, y los objetos que se definan con la clase *Contratista* utilizarán el método que corresponde a su respectiva clase.

En otras palabras, se dice que el polimorfismo le permite al programador, que herede clases a partir de una clase base, redefinir los miembros que le sean convenientes con versiones que se ajusten a las nuevas clases.

Existen dos formas de redefinir un miembro de una clase en una clase derivada. El primero consiste en implementar un miembro en la clase base en forma corriente,

```
public tipo NombreMiembro
{
      // Implementación en la clase base
}
```

y redefinirlo en la clase derivada utilizando el operador **new**. Así,

```
public new tipo NombreMiembro
{
      // Implementación en la clase derivada
}
```

Pero la forma más utilizada en C#, es aquella donde el programador de una clase base prevé cuales miembros se podrían eventualmente redefinir y los marca como *virtuales*. Para ello se utiliza la palabra clave **virtual**, así

```
public virtual tipo NombreMiembro
{
      // Implementación en la clase base
}
```

Cuando se hace de esta forma, en el momento de sobrescribir el miembro en la clase derivada se debe utilizar la palabra clave **override**, así

```
public override tipo NombreMiembro
{
      // Implementación en la clase derivada
}
```

El polimorfismo se puede observar en otros campos de la programación, incluso en los lenguajes que no pertenecen a la familia orientada a objetos. Generalmente, este tipo de polimorfismo se conoce como *sobrecarga* y lo abordaremos en las siguientes secciones. En C#, un buen ejemplo de sobrecarga es la que presenta el operador +, que puede aplicarse a valores numéricos o a cadenas de texto. Cuando se aplica a valores numéricos actúa como operación aritmética de suma, y cuando se aplica a cadenas de texto actúa como concatenación (unión) de cadenas. El operador + se adapta automáticamente al tipo de operandos sobre los cuales debe actuar. De igual manera también, los métodos pueden ser sobrecargados para que trabajen con argumentos de diferente tipo o incluso para que los mismos argumentos tengan uno y otro tipo.

```
string union = "ab" + "c";

int suma = 23 + 15;
```

Fig. 3.11: Sobrecarga del operador +. En *union* se almacena la cadena "abc" y en *suma* el valor 38.

La máxima potencia del polimorfismo se observa cuando actúa junto a otros elementos de la programación orientada a objetos que aún no se han descrito. Por ahora es importante asimilar y tener bien claro el concepto general para luego poder implantarlo bajo otros esquemas y comprender rápidamente su funcionalidad.

Ejemplo 3.8 Salario polimórfico

En seguida vamos a implementar la clase Empleado, descrita en esta sección, y a partir de ella se heredaran otras clases que redefinirán el método que permite calcular el salario de los trabajadores de una empresa.

Supongamos que, en general el salario de un empleado se calcula restando al salario básico mensual, el valor de la seguridad social que debe aportar un trabajador cualquiera. Si el método *CalcularSalario* se va a establecer como redefinible por las clases derivadas, se debe implementar como *virtual*, de la siguiente forma:

```
public virtual void CalcularSalario()
{
      double salario;
      double social;
      string mensaje;

      social = salarioBasico * seguridadSocial / 100;
      salario = basico - social;

      mensaje = "Seguridad social: " + social;
      mensaje += "\nSalario devengado: " + salario;

      MessageBox.Show(mensaje, "Empleado");
}
```

Pero se necesita una clase que se encargue de calcular el salario de un contratista, a quien se le paga de acuerdo al número de horas laboradas en el mes. En este caso el método antes definido no sirve. Una posible solución sería volver a comenzar desde cero e implementar una nueva clase con un método totalmente diferente al anterior. Pero una solución de este tipo desaprovecharía el código creado y nos obligaría a volver a programar algo que ya se ha realizado con anterioridad.

La solución orientada a objetos más correcta consiste en derivar una clase de Empleado y redefinir el método *CalcularSalario*. Este método se encargaría de establecer el salario devengado con base en el número de horas laboradas.

```
public override void CalcularSalario()
```

```
        {
            double salario;
            double valorHora;
            string mensaje;

            valorHora = base.SalarioBasico / 160;
            salario = valorHora * numeroHoras;

            mensaje = "Valor por hora: " + valorHora;
            mensaje += "\nSalario devengado: " + salario;

            MessageBox.Show(mensaje, "Contratista");
        }
```

En este último método existe un detalle especial. Se ha utilizado la palabra clave **base** para tener acceso a la propiedad *SalarioBasico* de la clase base. Esta es la forma que más se utiliza para hacer referencia a los miembros de una clase base desde las clases derivadas. También se puede acceder utilizando el nombre de la clase en vez de la palabra clave, pero es preferible utilizar la primera forma.

El componente, que incluye las clases, queda como sigue:

```
/* Archivo: Trabajador.cs */
using System;
using System.Windows.Forms;

public class Empleado
{
      string nombres;
      string apellidos;
      double salarioBasico;
      double seguridadSocial;

      public string Nombres
      {
            get { return nombres; }
            set { nombres = value; }
      }

      public string Apellidos
      {
            get { return apellidos; }
            set { apellidos = value; }
      }

      public double SalarioBasico
      {
            get { return salarioBasico; }
            set { salarioBasico = value; }
      }

      public double SeguridadSocial
      {
            set { seguridadSocial = value; }
      }

      // Método para calcular el salario devengado
      public virtual void CalcularSalario()
      {
            double salario;
            double social;
```

```
            string mensaje;
            // Calcular la seguridad social
            social = salarioBasico * seguridadSocial / 100;
            // Determinar el salario devengado
            salario = salarioBasico - social;
            // Construir el mensaje que se va a mostrar
            mensaje = "Seguridad social: " + social;
            mensaje += "\nSalario devengado: " + salario;

            MessageBox.Show(mensaje, "Empleado");
      }
}

public class Contratista : Empleado
{
      double numeroHoras;
      public double NumeroHoras
      {
            set { numeroHoras = value; }
      }

      // Reescritura del método CalcularSalario()
      public override void CalcularSalario()
      {
            double salario;
            double valorHora;
            string mensaje;
            // Calcular el valor de la hora con base en el salario básico
            valorHora = base.SalarioBasico / 160;
            // Determinar el salario devengado
            salario = valorHora * numeroHoras;
            // Construir el mensaje que se va a mostrar
            mensaje = "Valor por hora: " + valorHora;
            mensaje += "\nSalario devengado: " + salario;

            MessageBox.Show(mensaje, "Contratista");
      }
}
```

Compile este archivo en un ensamblado de tipo librería de enlace dinámico, *Trabajador.dll*, mediante la instrucción,

```
> csc /t:library Trabajador.cs
```

En seguida desarrollamos el programa que hace uso del ensamblado que acabamos de compilar, y de las clases que contiene.

```
/* Archivo: Ejemplo38.cs */
using System;

public class SalarioDevengado
{
      static void Main()
      {
            Empleado trabajador = new Empleado();
            trabajador.Nombres = "Juan";
            trabajador.Apellidos = "Pérez";
            trabajador.SalarioBasico = 450000;
            trabajador.SeguridadSocial = 16;
            trabajador.CalcularSalario();
```

```
            Contratista obrero = new Contratista();
            obrero.Nombres = "José";
            obrero.Apellidos = "Rodríguez";
            obrero.SalarioBasico = 450000;
            obrero.SeguridadSocial = 0;
            obrero.NumeroHoras = 80;
            obrero.CalcularSalario();
      }
}
```

Compile el archivo en un archivo ejecutable, con la instrucción

```
< csc /r:Trabajador.dll Ejemplo38.cs
```

En la ejecución del programa, resultado de nuestro ejemplo, podemos ver como el objeto creado a partir de la clase Empleado ejecuta la versión del método *CalcularSalario()* que esta clase implementa, y en su orden, el objeto de la clase Contratista, a pesar de heredar de la anterior, ejecuta su propia versión del método.

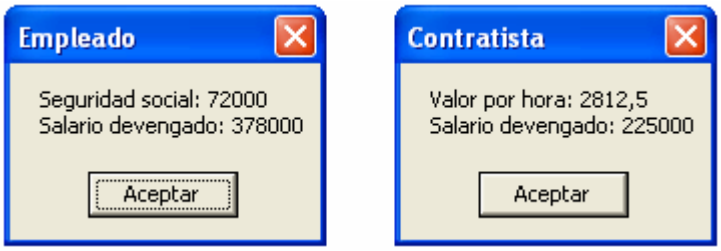

Figura 3.12: Ejecución del método *CalcularSalario()* del programa ejemplo38.exe. Cada clase ejecuta su propia versión del método, adquiriendo este un comportamiento polimórfico.

Sin embargo, aunque este es un ejemplo bastante esclarecedor sobre el tema, esta aún no es la máxima expresión del polimorfismo como tal. Tal vez el lector haya leído en alguna parte o escuchado en alguna conferencia, de esas que acostumbran a dar los expertos en programación orientada a objetos, que *el gato hereda de la clase mamífero* y que por lo tanto *todo gato es un mamífero* y como consecuencia de este raciocinio lógico cualquier gato puede ser estudiado desde el campo de los mamíferos. Es posible que este ejemplo *biológico computacional* haya sido suficiente para que muchos pudieran sacar la máxima ventaja de la herencia y el polimorfismo en el desarrollo de programas de computador. Pero en el caso de quién escribe estas líneas, un simple mortal como muchas deficiencias de comprensión, la cosa no fue una tarea tan fácil de asimilar. En el siguiente ejemplo, no con gatos ni mamíferos, sino con contratistas y empleados, vamos a ampliar la utilidad de las clases Empleado y Contratista para ver el polimorfismo en su máxima expresión y poder comprender mejor este intrincado raciocinio.

Ejemplo 3.9 Todo contratista es un empleado

En el siguiente ejemplo se utilizará el ensamblado con las clases desarrolladas en el ejemplo anterior y vamos a utilizarlas en un programa que manipula a un objeto del tipo Contratista a través de un objeto Empleado.

Sabemos que la clase Contratista es derivada de la clase Empleado. Por lo tanto podemos concluir que todo objeto definido con la clase Contratista se puede considerar como un objeto Empleado. Supongamos que se definen el objeto trabajador de la clase *Empleado* y el objeto obrero de la clase *Contratista*.

```
Empleado trabajador;
Contratista obrero;
```

A pesar de que un objeto de tipo *Empleado* tiene una estructura más reducida que un objeto de tipo *Contratista*, dada la relación de jerarquía existente entre las dos clases que los definen, es posible asignar al objeto *trabajador* un objeto de tipo Contratista realizando una *conversión forzada* de este último, así,

```
trabajador = (Empleado)obrero;
```

Realizada la conversión, se procede a manipular únicamente al objeto `trabajador`, que es de tipo Empleado. La pregunta que surge es, ¿Cómo se ejecutará el método *CalcularSalario()*? Compile y ejecute el siguiente programa para encontrar la respuesta:

```
/* Archivo: Ejemplo39.cs */

using System;

public class SalarioDevengado
{
      static void Main()
      {
            Empleado trabajador;
            Contratista obrero = new Contratista();
            obrero.NumeroHoras = 100;
            // Aquí se asume que el Contratista es un Empleado
            trabajador = (Empleado)obrero;
            // Datos del empleado
            trabajador.Nombres = "José";
            trabajador.Apellidos = "Rodríguez";
            trabajador.SalarioBasico = 450000;
            trabajador.SeguridadSocial = 0;
            // El empleado calcula su salario
            trabajador.CalcularSalario();
      }
}
```

Compile el archivo fuente mediante la instrucción,

```
> csc /r:Trabajador.dll Ejercicio39.cs
```

Al ejecutar el programa resultante observamos que, aunque el llamado al método `CalcularSalario()` lo hace un objeto de tipo *Empleado*, se ejecuta la versión que corresponde a los objetos de tipo *Contratista*.

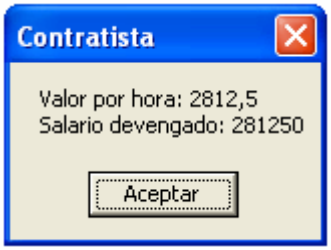

Figura 3.13: Ejecución del método *CalcularSalario()* por parte de un objeto de tipo Empleado que ha asumido el papel de un objeto de tipo Contratista.

En esencia, un objeto de tipo Empleado, que asume el papel de un objeto de tipo Contratista, ejecutará la operación acorde al objeto que representa. Este es el punto clave del polimorfismo ya que en un momento dado le permite al programador delegar la ejecución de algunas operaciones pertenecientes a objetos de diferentes tipos, en objetos de alta jerarquía que se encargarán de seleccionar automáticamente la operación apropiada al tipo de objeto que representen en un instante determinado.

En esta sección hemos descrito las cinco propiedades fundamentales de la programación orientada a objetos (abstracción, encapsulamiento, modularidad, herencia y polimorfismo), las cuales deben ser manejadas con absoluta claridad por parte del programador. Aunque cada una de estas propiedades, por cuestiones de organización, se ha descrito en forma separada, en la práctica todas convergen para constituir metodologías y técnicas, de desarrollo de componentes de software, altamente eficientes, y de las cuales el programador pueda sacar el máximo provecho posible.

Las anteriores descripciones se han realizado para las clases, que aunque son el elemento más importante de esta metodología de programación, no son las únicas que la conforman. Al menos existen dos elementos más que juegan un papel trascendental en la programación con C#, son las *interfaces* y las *colecciones.*

Interfaces

Las interfaces son módulos que solo definen un conjunto de métodos, pero no proporcionan implementación alguna. Su definición se realiza siguiendo la sintaxis,

```
interface NombreInterfase
{
      // Listado de firmas de propiedades y métodos
}
```

Una interfaz solo contiene las firmas de métodos y propiedades que serán implementados por las clases que la implementen.

Supongamos que se tiene la interfaz IPagos,

```
interface IPagos
{
      double SalarioBasico{get; set;}
      double SeguridadSocial{get; set;}
}
```

Para indicar que una clase *Nomina* se encargará de implementar esta interfaz se utiliza el operador dos puntos (:), de la misma forma como se establece la herencia,

```
class Nomina : IPagos
{
      public double SalarioBasico
      {
            // Implementación del método
      }

      public double SeguridadSocial
      {
            // Implementación del método
      }
}
```

Las interfaces son un elemento que puede ser de gran utilidad a la hora de dar comportamiento polimórfico a una implementación. Sin embargo, esta es una cualidad poco explotada por la gran mayoría de programadores, quienes únicamente ven a las interfaces como elementos que ayudan a forzar la estructura que deben poseer una o varias clases. Este aspecto se origina por la ayuda que brindan los sistemas de programación cuando generan código automáticamente. Cuando una clase implementa una interfaz, el sistema de programación genera automáticamente el código base de los métodos y propiedades que se deben implementar, y muchos creen que esta es la razón principal para la cual han sido definidas las interfaces. Pero el lector debe saber que esto es lo menos importante del trabajo con interfaces.

Para comprender mejor esta característica, vamos a romper las reglas establecidas con anterioridad, y describiremos primeramente un ejemplo didáctico con objetos del mundo real como son los aparatos eléctricos. El lector se ha preguntado, ¿por que razón, cuando adquiere un nuevo aparato electrónico, como puede ser un televisor, un computador o un equipo de sonido, pocas veces se preocupa si podrá conectar a las tomas de corriente que existen en su casa? Es más, las personas sabemos con absoluta precisión que esto no tendrá mayor problema. La razón es muy simple. Todos los aparatos electrónicos que se venden en el país vienen con un enchufe que calza con total precisión en las tomas de corriente de nuestras casas. Ante esto, un ingeniero electrónico simplemente dirá que los aparatos electrónicos vienen dotados de un enchufe que posee una interfaz de conexión de corriente eléctrica que cumple con las normas de estandarización que para el efecto se han aprobado en el país, o incluso en todo el mundo.

Estamos seguros que un enchufe eléctrico podrá conectarse en cualquiera de las tomas eléctricas de nuestra casa sin importar la marca o el material del cual están elaborados, por que la interfaz que los dos elementos exponen al mundo exterior es la misma para todos. Este es el aspecto clave. La interfaz es quien permite establecer un punto de conexión común a diversos dispositivos. Es decir que todos los componentes que expongan una misma interfaz calzan sin mayor problema cuando haya necesidad de realizar conexiones entre ellos.

Para aterrizar un poco en nuestra complicada exposición de carácter didáctico supongamos que existen dos clases, *Bancos* y *Nomina*, que se implementan de la siguiente forma:

```
public class Bancos
{
      double salarioBasico;
      double seguridadSocial;
      string sucursal;

      public Bancos(){}

      public double SalarioBasico
      {
            get {return salarioBasico;}
            set {salarioBasico = value;}
      }

      public double SeguridadSocial
      {
            get {return seguridadSocial;}
            set {seguridadSocial = value;}
      }

      public string Sucursal
      {
```

```
            get {return sucursal;}
            set {sucursal = value;}
      }
}

public class Nomina
{
      double salarioBasico;
      double seguridadSocial;
      int numeroNomina;

      public Nomina(){}

      public double SalarioBasico
      {
            get {return salarioBasico;}
            set {salarioBasico = value;}
      }

      public double SeguridadSocial
      {
            get {return seguridadSocial;}
            set {seguridadSocial = value;}
      }

      public int NumeroNomina
      {
            get {return numeroNomina;}
            set {numeroNomina = value;}
      }
}
```

Para no complicar las cosas, vamos a despreocuparnos de la herencia que pudimos haber desarrollado y solo utilizaremos las clases tal como se muestran aquí. Analice con detenimiento estas clases y suponga que ellas contienen datos de valores que son los totales provenientes de diferentes cuentas contables.

Ahora definimos una tercera clase que va a utilizar la información contenida en objetos que son instancias de estas clases, para sumar los valores de salario básico y seguridad social, y con ellos calcular un valor total. En esta tercera clase, llamada *Cuentas*, podría definirse un método llamado `Total` cuya estructura sería tal como se muestra en seguida:

```
public class Cuenta
{
      public Cuenta(){}

      public double Total(Bancos banco)
      {
            double total;
            total = banco.SalarioBasico
                        + banco.SeguridadSocial;
            return total;
      }

      public double Total(Nomina nomina)
      {
            double total;
            total = nomina.SalarioBasico
                        + nomina.SeguridadSocial;
            return total;
```

```
        }
}
```

Evidentemente las cosas funcionan sin mayor problema, incluso se hace un uso muy elegante de la sobrecarga, pero existen al menos dos problemas: el primero y más obvio es que ha sido necesario programar en la clase *Cuenta* dos métodos, algo así como dos tomas de corriente, que hacen exactamente lo mismo, y que se deben definir así para poder trabajar con cada una de las clases de objetos que se tienen; y el segundo problema tiene que ver con la perdida de modularidad de la clase *Cuenta*. Este último aspecto se hace evidente cuando el programador tenga más clases similares a *Bancos* y *Nomina*, y desee procesar sus objetos por medio del mismo método, `Total()`. Por ejemplo, supongamos que se tiene una clase *Caja*, que se implementa en forma similar a las anteriores,

```
public class Caja
{
        double salarioBasico;
        double seguridadSocial;

        public Caja(){}

        public double SalarioBasico
        {
                get {return salarioBasico;}
                set {salarioBasico = value;}
        }

        public double SeguridadSocial
        {
                get {return seguridadSocial;}
                set {seguridadSocial = value;}
        }
}
```

Para procesar los objetos de tipo *Caja* con la clase *Cuenta* es necesario modificar esta última clase, incluyendo un nuevo método `Total`, un tercer tipo de toma corriente, que sobrecargue a los ya existentes. Y así debería hacerse cada que sea necesario procesar objetos que, aunque similares, están definidos por clases diferentes.

De principio podría pensarse que situaciones como las descritas aquí son poco comunes en el desarrollo de aplicaciones de software. Pero se engaña quien piense de esta manera, o simplemente no tiene mucha experiencia en el desarrollo profesional de programas de computador. Cuando los programas crecen y la cantidad de clases que los componen también, son muchas las situaciones donde es deseable procesar un grupo de objetos de diferentes clases a través de un mismo proceso para ganar un mayor nivel de modularidad que será muy benéfico a la hora de hacer mantenimiento y modificaciones posteriores. La solución no es novedosa, incluso ya esta incorporada en las tomas y los enchufes de corriente eléctrica, y se llama interfaz.

Lo que se debe hacer para evitar tener que definir un nuevo método que hace lo mismo con objetos diferentes, es establecer una interfaz que implementaran todas las clases cuyos objetos se necesiten procesar con él. Así, teniendo en cuenta el problema que estamos tratando, bien podríamos definir una interfaz como la siguiente,

```
public interface IPagos
{
        double SalarioBasico{get; set;}
        double SeguridadSocial{get; set;}
```

```
}
```

Ahora se redefinen las clases en conflicto de tal manera que cada una de ellas implemente la interfaz IPagos. Así:

```
public class Bancos : IPagos
{
      double salarioBasico;
      double seguridadSocial;
      string sucursal;

      public Bancos()
      {
      }

      public double SalarioBasico
      {
            get {return salarioBasico;}
            set {salarioBasico = value;}
      }

      public double SeguridadSocial
      {
            get {return seguridadSocial;}
            set {seguridadSocial = value;}
      }

      public string Sucursal
      {
            get {return sucursal;}
            set {sucursal = value;}
      }
}

public class Nomina : IPagos
{
      double salarioBasico;
      double seguridadSocial;
      int numeroNomina;

      public Nomina()
      {
      }

      public double SalarioBasico
      {
            get {return salarioBasico;}
            set {salarioBasico = value;}
      }

      public double SeguridadSocial
      {
            get {return seguridadSocial;}
            set {seguridadSocial = value;}
      }

      public int NumeroNomina
      {
            get {return numeroNomina;}
            set {numeroNomina = value;}
      }
```

```
}

public class Caja : IPagos
{
      double salarioBasico;
      double seguridadSocial;

      public Caja(){}

      public double SalarioBasico
      {
            get {return salarioBasico;}
            set {salarioBasico = value;}
      }

      public double SeguridadSocial
      {
            get {return seguridadSocial;}
            set {seguridadSocial = value;}
      }
}
```

Ahora, nuestra clase *Cuenta* ya no necesita definir tres tomas de corriente para enchufar los objetos de *Bancos*, *Nomina* o *Caja*. Lo único que se debe hacer es implementar un método que acepte en su entrada objetos que cumplan con la interfaz *IPagos*. Para ello, el método `Total` debe aceptar como argumento un objeto de interfaz *IPagos*,

```
public class Cuenta
{
      public Cuenta(){}

      public double Total(IPagos pagos)
      {
            double total;
            total = pagos.SalarioBasico +
                        pagos.SeguridadSocial;
            return total;
      }
}
```

De esta manera, el método Total ha adquirido un comportamiento polimórfico al aceptar objetos de diferente tipo.

Ejemplo 3.10 Objetos con una misma interfaz

El siguiente ejemplo hace uso de las clases que se han analizado en esta sección y muestra la forma de utilizar los objetos creados con ellas. Primero, cree un ensamblado de tipo DLL que incluya la interfaz IPagos y la clase Cuenta, y guarde su código fuente en el archivo Cuentas.cs:

```
/* Archivo: Cuentas.cs */
using System;

public interface IPagos
{
      double SalarioBasico{get; set;}
      double SeguridadSocial{get; set;}
}
```

```
public class Cuenta
{
      public Cuenta(){}

      public double Total(IPagos pagos)
      {
            double total;
            total = pagos.SalarioBasico + pagos.SeguridadSocial;
            return total;
      }
}
```

Compile este archivo con la instrucción,

```
> csc /t:library Cuentas.cs
```

Las clases Bancos y Nomina, se incluirán en el archivo *Ejemplo310.cs*, al igual que la clase que contiene el método *Main*:

```
/* Archivo: Ejemplo310.cs */

using System;

public class Bancos : IPagos
{
      double salarioBasico;
      double seguridadSocial;
      string sucursal;

      public Bancos(){}

      public double SalarioBasico
      {
            get {return salarioBasico;}
            set {salarioBasico = value;}
      }

      public double SeguridadSocial
      {
            get {return seguridadSocial;}
            set {seguridadSocial = value;}
      }

      public string Sucursal
      {
            get {return sucursal;}
            set {sucursal = value;}
      }
}

public class Nomina : IPagos
{
      double salarioBasico;
      double seguridadSocial;
      int numeroNomina;

      public Nomina(){}
```

```
      public double SalarioBasico
      {
            get {return salarioBasico;}
            set {salarioBasico = value;}
      }

      public double SeguridadSocial
      {
            get {return seguridadSocial;}
            set {seguridadSocial = value;}
      }

      public int NumeroNomina
      {
            get {return numeroNomina;}
            set {numeroNomina = value;}
      }
}

class CuentasContables
{
      public static void Main(string[] args)
      {
            Bancos banco = new Bancos();
            banco.SalarioBasico = 1000;
            banco.SeguridadSocial = 200;

            // Un objeto se puede definir a partir de la interfaz
            IPagos nomina = new Nomina();
            nomina.SalarioBasico = 500;
            nomina.SeguridadSocial = 60;

            Cuenta cuenta = new Cuenta();
            double total1 = cuenta.Total(banco);
            double total2 = cuenta.Total(nomina);

            Console.WriteLine("Total1 = " + total1.ToString());
            Console.WriteLine("Total2 = " + total2.ToString());

            Console.Write("Presione una tecla para continuar. . . ");
            Console.ReadKey(true);
      }
}
```

Para compilar este archivo se debe utilizar la instrucción

```
> csc /r:Cuentas.dll Ejercicio310.cs
```

En general, todas las clases que implementen la interfaz *IPagos* podrán ser procesadas por medio del método `Total` de la clase *Cuenta*. Esto trae una ventaja directa, y es que reduce la cantidad de código que se debe programar para realizar una determinada tarea. Además, podrían incluirse en un arreglo o una colección diversos objetos y luego ser procesados por un único método.

Colecciones

Una colección es una forma de agrupar objetos que pueden ser del mismo tipo o de tipos diferentes. Estas agrupaciones le permiten al programador establecer algoritmos

que actúen sobre todos la colección sin necesidad de preocuparse sobre las características particulares de uno y otro objeto.

Por ahora no ahondaremos más en este tema, y su estudio en profundidad lo estaremos abordando en un capítulo posterior.