Capítulo 3
Programación orientada a objetos

1. Principios de la programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma muy extendido en desarrollo de software. Viene a completar un panorama que ya es muy rico del paradigma de procedimientos, así como del funcional.

La POO es una forma de diseño de código que aspira a representar los datos y las acciones como si formaran parte de clases; ellas mismas se convierten en objetos durante su creación en memoria. Este concepto se ha presentado rápidamente en el capítulo anterior: ahora es el momento de comprender su funcionamiento de manera más detallada.

1.1 ¿Qué es una clase?

Una clase es un elemento del sistema que forma la aplicación. Una clase contiene dos tipos de elementos de código: datos y métodos (que representan acciones). Hay que ver la clase como una caja donde es posible ordenar estos dos tipos de elementos. Para hacer un paralelismo con la vida real, podemos comprender fácilmente que la definición de una clase se aplica a un objeto como un ordenador, por ejemplo. Este último dispone de métodos (encender, apagar, etc.), así como de propiedades (número de pantallas, cantidad de RAM, etc.).

De manera conceptual, una clase solo es una definición. Una vez que haya decidido lo que debe contener, así como sus métodos, es conveniente crearla. Esta acción se llama instanciación. Tras esta operación, obtenemos una instancia en memoria de un objeto.

Vamos a intentar hacer una comparación. Tomemos el ejemplo de una fábrica de producción de objetos de madera. Para poder crear un objeto, se necesita un plan (la clase). Gracias a este último, la máquina puede cortar y juntar los diversos elementos (datos y métodos) para crear una instancia nueva (instanciación).

En C#, la declaración de una clase se hace con la palabra clave `class`. Hay algunas posibles particularidades, especialmente el ámbito, que estudiaremos justo después, en la sección ¿Qué se puede declarar dentro de una clase? - Métodos, así como los conceptos de `static`, `sealed` y el de `partial`. La sintaxis completa de la declaración de una clase es la siguiente:

```
AMBITO [static] [sealed] [partial] class NOMBRE_CLASE
```

El nombre de la clase es libre, pero debe cumplir dos normas:

– Solo puede contener caracteres alfanuméricos y el carácter guion bajo («_»).

– No puede empezar con un número.

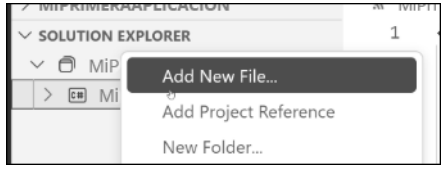
Además de estas normas, es frecuente que los desarrolladores de C# respeten una convención de sintaxis: el uso de PascalCase. Esto indica que el nombre empieza con una mayúscula y cada palabra también empieza con una mayúscula, por ejemplo: `OrdenadorPortatil`. El lenguaje y el compilador no prohíben escribir `ordenadorPortatil`, `Ordenadorportatil` o incluso `ordenadorportatil`, pero estas distintas declaraciones no respetan la convención ampliamente aceptada y aplicada. Finalmente, aunque sea posible, se recomienda evitar caracteres acentuados en el nombre de una clase. Por ejemplo, es mejor llamar a su clase `Peaton` en lugar de `Peatón`, para que el código C# producido sea lo más parecido posible al que tendríamos en inglés.

En el programa de base creado en C# en el capítulo anterior, se ha creado una clase `Program` de manera predeterminada. Podemos constatar que no hay concepto de ámbito ni de `partial` o `static`. Después de declararla, la clase define un bloque donde podemos implementar los datos y los métodos que necesita nuestro programa para funcionar.

1.1.1 Las clases en Visual Studio Code

Para crear una clase en Visual Studio Code, hay que seguir las etapas que aparecen a continuación:

- Expanda la vista **Solution Explorer** del proyecto.
- Colóquese en la carpeta donde desea crear la nueva clase (o directamente en el nombre del proyecto si desea crearla en la raíz).
- Haga clic derecho para seleccionar el elemento de menú **Add a New File**.
- Escriba el nombre de la clase en la pequeña ventana emergente que se abre en la parte superior central de la pantalla (siempre sin espacios ni caracteres especiales).



Añadir una clase nueva con Visual Studio Code

Después de estas operaciones, en la jerarquía situada a la izquierda hay disponible un archivo nuevo que lleva el nombre de la clase seguido por la extensión .cs. De manera predeterminada, este archivo estará abierto.

1.1.2 Herencia

Hay un concepto extremadamente importante en POO: la herencia. En general, si tiene la posibilidad de decir «X es una Y», el equivalente podría ser decir «X hereda de Y». X toma todas las propiedades y comportamientos de Y, pero los particulariza. Vamos a dar un ejemplo concreto: «Un Mac es un ordenador». Entonces, a nivel del desarrollo orientado a objetos, un Mac toma todas las propiedades y comportamientos de un ordenador, pero los particulariza aportando sus propios elementos. Decimos en este caso que `Mac` es una clase hija de la clase `Ordenador`.

En C#, este concepto es central porque todos los elementos que va a manipular obviamente heredan de la clase `System.Object`, que define el comportamiento básico de cualquier objeto. Además, a diferencia de otros lenguajes (como C++), en C# no es posible heredar de varias clases: solo es posible tener una clase madre. Si no se especifica ninguna clase madre, es por definición la clase `System.Object` la que constituye la clase madre (sin que se requiera ninguna operación).

Observación

En C# no se puede heredar de varias clases. Por eso hay que elegir la clase de la que se hereda. Si no se especifica nada, el compilador genera automáticamente, de manera transparente, una herencia de la clase `System.Object`, como se describe con anterioridad. Si especificamos una herencia, eso no quiere decir que la clase herede de `System.Object` y de la clase heredada, sino solo de la clase heredada que sustituye a la herencia generada por el compilador. La clase heredada, en sí misma, hereda de otra clase o directamente de `System.Object`. En última instancia, todas las clases en C# heredan de `System.Object` de una forma u otra.

Para indicar que una clase hereda de otra, hay que usar los dos puntos seguidos de la clase de la que se quiere heredar:

```
class Ordenador { }
class Mac : Ordenador { }
```

Por supuesto, el hecho de que una clase herede de otra no quiere decir que por fuerza tenga acceso a todo lo que se ha definido dentro de la clase madre.

1.1.3 Encapsulación

Todo lo que se encuentra en el interior de una clase se designa mediante un término muy específico: encapsulación. Con ella también aparece el concepto de ámbito, que indica cómo se perciben las cosas desde un punto de vista exterior a la clase.

El ámbito permite definir la visibilidad de un elemento de una clase o de la misma clase. En total hay siete ámbitos en C#:

- `public`: define que el elemento es completamente visible dentro y fuera de la clase.
- `private`: define que el elemento solo es visible en el interior de la clase donde se ha declarado, mientras que es completamente invisible desde el exterior.
- `internal`: define que el elemento solo es visible dentro del proyecto donde se ha declarado. Podemos considerarlo como `public`, pero solo dentro del proyecto donde fue declarado. Otro proyecto que referencie nuestro proyecto no tiene conocimiento de un elemento que se ha declarado como `internal`. De manera predeterminada, si no explica el ámbito explícito en una clase, el compilador selecciona `internal`.
- `protected`: define que el elemento solo es visible en el interior de la clase donde se ha declarado y dentro de su jerarquía de clases hijas. Eso se une con el concepto de la herencia, que veremos más adelante en este capítulo.
- `protected internal`: define una suma entre `protected` e `internal`. Un elemento declarado con este ámbito es visible por la clase interesada, sus clases hijas, así como por todas las otras clases dentro del mismo proyecto. Esto también significa que, si se declara una clase hija fuera del proyecto actual, puede tener acceso a un elemento `protected internal`, al igual que cualquier clase del mismo proyecto.
- `private protected`: define una intersección entre `protected` e `internal`. Un elemento declarado con este ámbito solo es visible por la clase interesada y por sus clases hijas definidas dentro del mismo proyecto. Esto quiere decir que una clase hija definida fuera del proyecto actual no podrá acceder a este elemento.

– `file`: agregado en C# 11, este ámbito define la visibilidad solo dentro del marco del archivo actual. Este ámbito es muy particular porque no está pensado para que lo utilicen directamente los desarrolladores. Existe, principalmente para herramientas de generación automática de código. Sin embargo, en casos muy raros puede resultar útil declarar un elemento que solo existe como parte de un archivo para un algoritmo específico. También cabe señalar que, a diferencia de otros ámbitos, este ámbito solo es válido para la declaración de un tipo. No se puede aplicar a un método, campo o propiedad.

Con todos estos ámbitos, se puede crear la clase que corresponde con precisión a las necesidades de nuestra aplicación, para evitar que ciertos elementos no salgan del perímetro de la clase. Retomando nuestro ejemplo, consideramos que la clase `Ordenador` dispone de un booleano que indica si la máquina está encendida o no. Para evitar que nadie pueda manipular este dato de forma directa, la manera de proceder es definirla como públicamente accesible en modo de lectura, pero privado en lo que respecta a la escritura. En consecuencia, solo un método público, definido en esa clase, como por ejemplo `Encender` o `Apagar`, puede cambiar el valor de este indicador. Así nos protegemos de un cambio de estado no controlado (porque podemos considerar que la operación de extinción necesita efectuar algunas operaciones con antelación antes de transferir el booleano).

1.2 ¿Qué se puede declarar dentro de una clase?

Como ya hemos visto, dentro de una clase podemos declarar dos tipos de elementos: métodos (acciones) y datos. Vamos a ver rápidamente cómo declararlos.

1.2.1 Métodos

Un método traduce una acción que se puede invocar en la clase. Durante la declaración de un método, hay que hacerse las siguientes preguntas:

– ¿Se trata de una acción que debe poder realizarse desde el exterior o solo desde el interior de la clase?

– ¿Se espera un valor de retorno particular?

– ¿Es necesaria alguna información para que este método funcione?

Ya ha tenido una vista previa de una llamada de método en el primer capítulo, en la clase `Consola`: `WriteLine` y `ReadLine`. Estos dos métodos ilustran los puntos antes citados:

– `WriteLine` se debe poder llamar desde el exterior. No esperamos que devuelva un valor después de llamarla, pero es necesario transmitirle la información que queremos escribir.
– `ReadLine` también se debe poder llamar desde el exterior. Necesitamos recuperar solo la información introducida por el usuario, sin que sea preciso transmitirle una información cualquiera.

La sintaxis de declaración de un método dentro de una clase es la siguiente:

```
ÁMBITO [static] TIPO_RETORNO NOMBRE_MÉTODO([PARÁMETROS])
```

El tipo de retorno debe corresponder a un tipo C# conocido. Por ejemplo, si queremos crear un método que realiza la suma de dos números y devuelve el resultado, todo accesible de manera pública, lo declaramos de la siguiente manera:

```
public int Addition(int primero, int segundo) {}
```

Observación

Cuando declaramos un método con un valor de retorno sin escribir el contenido del método, el compilador emite inmediatamente un error de compilación. Esto se debe al hecho de que es obligatorio que cada método que devuelve un resultado contenga una instrucción `return`.

Cuando un método debe devolver un valor, hay que usar la palabra clave `return` para definir el valor que deseamos devolver. La instrucción `return` se puede usar directamente con un valor o podemos utilizar una variable del tipo de retorno esperado. En el caso del ejemplo anterior, son válidas estas dos maneras de escribir el método:

```
public int Suma(int primero, int segundo)
{
    return primero + segundo;
}
public int Suma(int primero, int segundo)
{
```

```
    int resultado = primero + segundo;
    return resultado;
}
```

Un elemento importante que hay que a recordar: del mismo modo que hemos visto en el capítulo anterior con la declaración de clases del mismo nombre dentro del mismo espacio nombres, no se puede declarar dos veces el mismo método dentro de una misma clase. Si los nombres son idénticos y los parámetros también lo son, entonces el compilador C# considera que es el mismo método. El valor de retorno no constituye un elemento distintivo. Así, la declaración de los dos métodos siguientes dentro de la misma clase es imposible y eso provoca un error de compilación:

```
public int Suma (int primero, int segundo)
{
    return primero + segundo;
}
public void Suma (int primero, int segundo)
{
}
```

Observación

Como podemos comprobar en el ejemplo anterior, la palabra clave void especifica que el método no devuelve ningún resultado. El concepto de tipo de retorno es obligatorio y hay que usar esta palabra clave para indicar cuándo no lo hay.

Si el método no toma parámetros, la presencia de paréntesis que se abren y se cierran unidos al nombre del método es, a pesar de todo, necesaria para indicar que se trata de un método:

```
public void MiMetodo()
{
}
```

Dentro de un método que declara su propio bloque, se pueden declarar variables y constantes que se consideran únicamente locales (es decir, visibles dentro del método y de todos sus subbloques, pero invisibles dentro de los bloques padres, directos o indirectos).

Cabe señalar que incluso en el caso de que un método no devuelva ningún valor, es posible utilizar la declaración `return` para detener la ejecución del método. En este caso concreto bastará con escribir la palabra clave `return` y terminar con punto y coma. El código después de la declaración `return` no se ejecutará:

```
public void Suma (int primero, int segundo)
{
    return;
    // la siguiente línea de código nunca se ejecutará
    int resultado = primero + segundo;
}
```

1.2.2 Declarar un dato

Hay dos maneras de declarar un dato dentro de una clase: mediante una propiedad o mediante un campo. Las dos son completamente legítimas, pero no responden a las mismas necesidades.

El caso más sencillo es la declaración de un campo. Un campo de una clase se define de la siguiente manera:

```
ÁMBITO TIPO NOMBRE_DEL_MIEMBRO;
```

Observación

El concepto de ámbito no es obligatorio y es posible omitirlo. En ausencia de la definición de ámbito, es `private` el utilizado por un campo en el compilador. Sin embargo, es muy recomendable añadirla por motivos de claridad.

Por ejemplo, si en nuestra clase `Ordenador` queremos almacenar el año de compra bajo la forma de un entero accesible para todos, podemos crear un campo como este:

```
public int anoCompra;
```

Como podemos ver aquí arriba, la convención de sintaxis recomendada para la escritura de los campos es *lower camel casing*, precedido de un guion bajo. Esta convención indica que la primera letra es minúscula, pero todas las palabras siguientes empiezan por su propia mayúscula.

Capítulo 3
Introducción al lenguaje C#

1. La sintaxis

1.1 Los identificadores

Los identificadores son los nombres que se asignan a las clases y a sus miembros. Un identificador debe estar compuesto por una única palabra. Ésta debe empezar por una letra o un carácter underscore (_). Los identificadores pueden tener letras mayúsculas o minúsculas, pero como el lenguaje C# es sensible a mayúsculas/minúsculas, éstas se deben respetar para que la referencia al identificador sea correcta: `miIdentificador` es diferente a `MiIdentificador`.

1.2 Las palabras clave

Las palabras clave son los nombres que reserva el lenguaje C#. Se interpretan por el compilador y, por tanto, no se pueden usar como identificadores. Estas palabras clave se diferencian en el editor de texto de Visual Studio utilizando el color azul (con los argumentos de apariencia predeterminados).

Si necesita utilizar una palabra clave como un identificador para un miembro, es necesario utilizar un prefijo con el nombre del identificador: el carácter @. La siguiente sintaxis es errónea y el compilador no la podrá ejecutar:

```
private bool lock;
```

Si utilizamos el prefijo @ con el miembro `lock`, el compilador considera que es un identificador y no una palabra clave:

```
private bool @lock;
```

El carácter @ también puede servir como prefijo de los identificadores que no tienen ningún conflicto con las palabras claves, de esta manera `@miIdentificador` se interpretará de la misma manera que `miIdentificador`.

A continuación, se muestra una lista de las palabras clave del lenguaje C#. Se explicarán, en parte, a lo largo del libro:

abstract	add	as	ascending	async
await	base	bool	break	by
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	descending	do	double	dynamic
else	equals	enum	event	explicit
extern	false	file	finally	fixed
float	for	foreach	from	get
global	goto	group	if	implicit
in	int	interface	internal	into
is	join	let	lock	long
nameof	namespace	new	null	object
on	operator	orderby	out	override
params	partial	private	protected	public
readonly	ref	remove	required	return
sbyte	sealed	select	set	short

sizeof	stackalloc	statics	string	struct
switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe
ushort	using	value	var	virtual
volatile	void	where	while	yield

1.3 Las reglas de puntuación

El objetivo de las reglas de puntuación es separar las instrucciones del programa de manera lógica, comprensible por el ser humano e interpretable por el compilador.

Cualquier instrucción debe terminar con un punto y coma `;`. Si se olvida al final de la instrucción, el compilador devuelve un error de sintaxis. Sin embargo, la ventaja es poder escribir una instrucción en varias líneas:

```
int i = 5
        + 2;
```

El punto . después de un identificador permite acceder a los miembros de un objeto. A través de IntelliSense, Visual Studio muestra la lista de miembros disponibles, tan pronto como se añade el punto a un objeto:

```
miObjeto.miPropiedad
```

Las llaves `{` y `}` se usan para agrupar varias instrucciones dentro de un bloque de control o de un método. Indican dónde comienzan las instrucciones y dónde terminan:

```
class Program
{
}
```

Los paréntesis (y) se usan para declarar o para llamar a métodos. Pueden contener argumentos después de la declaración del método. Los argumentos de un método se separan con una coma `,`:

```
miObjeto.miMetodo(Parametro1, Parametro2);
```

Los paréntesis también se utilizan para agrupar las instrucciones de la misma manera que para una operación matemática.

Los corchetes `[` y `]` permiten acceder a los elementos de un array o, si la clase contiene una propiedad para indexar, a los elementos de una clase. Por ejemplo, si la clase `miObjeto` es un array de valores de tipo `string`, para acceder a su primer elemento, la sintaxis sería la siguiente:

```
string s = miObjeto[0];
```

Los elementos de los arrays se indexan comenzando por 0.

1.4 Los operadores

1.4.1 Los operadores de cálculo

Los operadores de cálculo permiten, como en matemáticas, realizar operaciones.

La adición se realiza con el operador `+`:

```
i = 5 + 2;          // i = 7
```

La sustracción se realiza con el operador `-`:

```
i = 5 - 2;          // i = 3
```

La multiplicación se realiza con el operador `*`:

```
i = 5 * 2;          // i = 10
```

La división se realiza con el operador `/`:

```
i = 6 / 2;          // i = 3
```

El modulo se realiza con el operador `%`:

```
i = 5 % 2;          // i = 1
```

1.4.2 Los operadores de asignación

Los operadores de asignación permiten asignar un valor a una variable. El operador más utilizado es el carácter =:

```
i = x;
```

También es posible realizar una asignación y un cálculo al mismo tiempo, combinando dos operadores:

```
i += 1;
```

usando el operador +=, hay una asignación a la variable de su propio valor, sumando el valor de la derecha del operador. Esta instrucción es equivalente a la siguiente:

```
i = i + 1;
```

La combinación de operadores de cálculo y de asignación es posible para todos los operadores:

```
int i = 5;
i += 2;          // i = 7
i -= 2;          // i = 5
i *= 2;          // i = 10
i /= 2;          // i = 5
i %= 2;          // i = 1
```

1.4.3 Los operadores de comparación

Los operadores de comparación se utilizan fundamentalmente para tomar decisiones dentro de instrucciones de control.

El operador == determina si dos variables son iguales:

```
x == y           // devuelve true si x igual a y
```

El operador != determina si dos variables son diferentes:

```
x != y           // devuelve true si x es diferente de y
```

El operador > determina si la variable de la izquierda es estrictamente superior a la variable de la derecha:

```
x > y            // devuelve true si x es superior a y
```

El operador `>=` determina si la variable de la izquierda es superior o igual a la variable de la derecha:

```
x >= y          // devuelve true si x es superior o igual a y
```

El operador `<` determina si la variable de la izquierda es estrictamente inferior a la variable de la derecha:

```
x < y           // devuelve true si x es inferior a y
```

El operador `<=` determina si la variable de la izquierda es inferior o igual a la variable de la derecha:

```
x <= y          // devuelve true si x es inferior o igual a y
```

El operador (y palabra clave) `is` permite determinar el tipo de un objeto:

```
x is int        // devuelve true si x es del tipo int
```

El filtrado por motivo permite verificar si un valor corresponde a un motivo. Se trata de utilizar el operador `is` para definir el motivo que puede reemplazarse en instrucciones condicionales:

```
o is DateTime d // devuelve true si o es de tipo DateTime
```

La variable `o` es automáticamente convertida en el tipo testeado y almacenada en la nueva variable `d`, que se puede utilizar de manera clásica.

También es posible combinar los operadores de comparación con los operadores lógicos. El operador `&&` permite especificar un AND lógico, mientras que el operador `||` especifica un OR lógico. Las diferentes expresiones se pueden combinar con ayuda de los paréntesis para modificar el orden de interpretación:

```
(x >= 0 || x > 10) && (x <= 1 || x < 25)
```

1.5 La declaración de variables

La declaración de variables se realiza especificando su tipo y posteriormente indicando su identificador. La siguiente instrucción declara una variable llamada `s` de tipo `string`:

```
string s;
```

La instrucción de declaración termina como cualquier instrucción, es decir, con un punto y coma.

Una variable se puede declarar e inicializar con la misma instrucción:

```
string s = "El valor de mi variable";
```

También es posible declarar e inicializar varias variables en una única instrucción, con la condición de que sean del mismo tipo. Las variables se separan con una coma:

```
bool b1 = true, b2 = false;
```

Una variable también se puede marcar con la palabra clave `const`, que especifica que el valor de la variable no se puede modificar durante la ejecución. Es una variable en modo de solo lectura:

```
const int i = 0;
```

Añadiendo la palabra clave `required` a una propiedad o a un campo, indica que los elementos que llaman al constructor tienen la obligación de inicializar estos valores.

```
required int i;
```

El ámbito de una variable declarada en el cuerpo de un método se limita a ese método, es decir, se elimina cuando finaliza la ejecución del método. Queda fuera de ámbito.

Las variables declaradas dentro de un bloque condicional o iterativo solo son accesibles dentro de este bloque. Para los bloques iterativos, la variable se elimina al final del bucle y se inicializa de nuevo en la siguiente iteración del bucle.

1.6 Las instrucciones de control

1.6.1 Las instrucciones condicionales

Las instrucciones condicionales permiten ejecutar una porción de código en función de comprobaciones realizadas sobre las variables de la aplicación. Existen dos tipos de estructuras condicionales: los bloques `if` y los bloques `switch`.

if, else y else if

Sintaxis general:

```
if (expresión)
{
       instrucciones
}
[else if (expresión)
{
       instrucciones
}]
[else
{
       instrucciones
}]
```

La instrucción `if` evalúa una expresión booleana y ejecuta el código si esta expresión es verdad (`true`). Su sintaxis es la siguiente:

```
if (x > 10)
{
   // Instrucciones que se ejecutan si x es superior a 10
}
```

La expresión a evaluar se inserta entre paréntesis después de la palabra clave `if` y debe tener un resultado de tipo booleano.

La instrucción `else` permite integrar el código que se ejecutará si la expresión que se evalúa en la instrucción `if` es falsa (`false`):

```
if (x > 10)
{
   // Instrucciones que se ejecutan si x es superior a 10
}
else
{
   // Instrucciones que se ejecutan si x es inferior o igual a 10
}
```

La instrucción `else if` permite evaluar una nueva expresión cuando el resultado de la instrucción `if` es falsa. Puede haber varias instrucciones `else if` pero un bloque de decisión `if` siempre debe empezar por una instrucción `if`, seguida de una o varias instrucciones `else if` y terminar de manera no obligatoria por una instrucción `else`:

```
if (x > 10)
{
   // Instrucciones que se ejecutan si x es superior a 10
}
else if (x = 10)
{
   // Instrucciones que se ejecutan si x es igual a 10
}
else
{
   // Instrucciones que se ejecutan si x es inferior a 10
}
```

Las instrucciones dentro de la estructura de decisión se encierran entre llaves. Esto indica el inicio y el fin del bloque. En el caso en el que el bloque solo tenga una línea, se permite omitir las llaves (también válido para las demás instrucciones de control):

```
if (x > 10)
   x -= 10;
else if (x = 10)
   x -= 1;
else
   x += 10;
```

El operador de condición `null ?.` permite comprobar el valor `null` cuando se accede a los miembros de una clase. Esto limita el número de instrucciones condicionales y el código resulta más legible. Por ejemplo, para acceder a una propiedad encapsulada en un objeto y asegurarse de que ningún padre es `null` y, por tanto, que no se producirá ninguna excepción durante su acceso, deberíamos escribir una instrucción de este tipo:

```
string MajorVersion;
if(soft!=null && soft.Version!=null)
{
    MajorVersion = soft.Version.Major;
}
```