

## TEMA 2. LENGUAJE DE PROGRAMACIÓN EN DISEÑO DE INTERFACES

# Principios de la programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma muy extendido en el 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. ¿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 (acciones). Hay que ver la clase como una caja donde se pueden ordenar estos dos tipos de elementos. Para hacer un paralelismo con la vida real, se puede 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, se obtiene 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 se estudiará justo después, en la sección ¿Qué se puede declarar dentro de una clase? - Métodos, así como el concepto de `static` y el de `partial`. La sintaxis completa de la declaración de una clase es la siguiente:

ÁMBITO [static] [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 admitida y aplicada.

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

## a. 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. Entonces podemos decir 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 nada, la clase System.Object es la clase madre (no se requiere ninguna operación).

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 se especifica 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.

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

.

## b. Encapsulación

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

El ámbito permite definir la visibilidad de un elemento de una clase o de la misma clase. En total hay seis 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 de un único proyecto. Otro proyecto que usa 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 y dentro del mismo proyecto. Esto también significa que, si se declara una clase hija fuera del proyecto actual, puede ver un elemento **protected internal**, al igual que cualquier clase del mismo proyecto puede acceder a ella.
- **Private protected:** define una intersección entre **protected** e **internal**. Un elemento declarado con este ámbito sólo 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.

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, lo definimos como públicamente accesible en modo de lectura, pero privado en lo que respecta a la escritura. Por ese motivo, solo un método público, como **Encender** o **Apagar**, puede cambiar el valor de este indicador. Así nos protegemos de un cambio de estado no controlado (porque se puede considerar que la operación de extinción necesita efectuar algunas operaciones con antelación antes de transferir el booleano).

## 2. ¿Qué se puede declarar dentro de una clase?

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

### a. 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 se debería poder realizar desde el exterior o solo desde el interior de la clase?
- ¿Se espera que devuelva un valor característico?
- ¿Se necesita determinada información para funcionar?

Ya ha tenido una vista previa de una llamada de método en el primer capítulo, en la clase `Console`: `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 se le transmite información que queremos escribir.
- `ReadLine` también se debe poder llamar desde el exterior. Necesitamos recuperar solo la información introducida por el usuario, sin necesidad de 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) {}
```

Cuando se declara un método con un valor de retorno sin escribir el contenido del método, el compilador emite 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 se desea devolver. La instrucción `return` se puede usar directamente con un valor o 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;
}

```

Es importante que recuerde: 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)
{
}

```

Como puede 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).

## **b. Declarar un dato**

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

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

ÁMBITO TIPO NOMBRE\_DEL\_MIEMBRO;

El concepto de ámbito no es obligatorio y es posible omitir. En ausencia de la definición de ámbito, se usa `private`. Sin embargo, es muy recomendable añadirla por motivos de claridad.

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

```
public int anoCompra;
```

Como se puede ver aquí arriba, la convención de sintaxis recomendada para la escritura de los miembros es `lower camel casing`. Esta convención indica que la primera letra es minúscula, pero todas las palabras siguientes empiezan por su propia mayúscula.

Una vez declarado, podemos acceder a nuestro miembro según su ámbito, desde los métodos de la clase o desde el exterior. La ventaja de este enfoque es que se trata de una variable global dentro de la clase, accesible para todos los métodos presentes en el interior. Sin embargo, un miembro no permite la distinción entre un derecho de lectura y de escritura. Cuando se declara un miembro con un ámbito dado, es accesible en modo de lectura y de escritura. Se puede limitar la escritura indicando que un miembro se puede definir solo durante la fase inicial de creación de la clase. Esta palabra clave es `readonly`. Así, si queremos que el año de compra solo se pueda introducir durante la instanciación de la clase, hay que escribir el siguiente código:

```
public readonly int anoCompra;
```

Para mitigar esta limitación, hay otra manera de almacenar datos dentro de una clase: las propiedades. Este concepto un poco particular corresponde a un dato que se expone mediante un método de lectura (`get`) y de escritura (`set`).

La sintaxis es la siguiente:

```
ÁMBITO TIPO NOMBRE_PROPRIEDAD { ÁMBITO get { } ÁMBITO set { } }
```

Este planteamiento tiene múltiples ventajas:

- Se puede definir un ámbito distinto para la lectura y la escritura. Se puede considerar fácilmente una propiedad que sea accesible de manera pública en modo de lectura, pero solo de manera privada en modo de escritura.
- `get` y `set` son métodos un poco particulares y, por eso, se puede escribir un cuerpo de método. Del mismo modo, con el fin de aumentar la productividad, se puede usar la propiedad automática, que veremos un poco más adelante.

El planteamiento de una propiedad, en un principio, es actuar como proxy hacia un miembro para poder guiar su encapsulación. Por ejemplo, vamos a imaginar que queremos transformar el año de compra definido arriba como dato únicamente accesible en modo de

lectura de forma pública. Inicialmente, hay que hacer que el miembro sea privado y, en una segunda fase, crear una propiedad que permite gestionar nuestras limitaciones:

```
private int anoCompra;  
public int AnoCompra  
{  
    get { return anoCompra; }  
    private set { anoCompra = value; }  
}
```

Analizamos este bloque de código para comprender su funcionamiento. La propiedad se ha definido de manera pública en el sentido global. Esto quiere decir que, en ausencia de ámbito en el get o el set, se aplicará la propiedad de manera predeterminada. También constatamos que la convención de sintaxis para una propiedad es la misma que para una clase: Pascal Casing.

Después, en el bloque definido mediante el get de una propiedad, se necesita una instrucción return que devuelve el valor deseado, del mismo tipo que la propiedad. Esto también permite realizar una transformación cuando se quiere devolver un dato de otro valor.

A continuación, viene el bloque definido mediante el set. Aquí, hemos decidido que la escritura debía permanecer privada, es decir, accesible solo desde los métodos de la clase. Para eso, hemos definido el ámbito, colocado justo antes de la palabra clave set. En este escenario no hay ninguna necesidad de devolver cualquier valor. Sin embargo, el set expone un parámetro particular, representado por la palabra clave value. Esta contiene el valor que se ha asignado a la propiedad, y que podemos asignar a nuestro miembro.

Esta propiedad es un ejemplo bastante clásico, y es completamente posible optimizar su escritura. Desde la versión 3 del lenguaje C#, existe el concepto de propiedad automática. Esto significa que, escribiendo solamente las palabras clave get y set (si hiciera falta, con sus ámbitos respectivos) sin darles cuerpo, el compilador C# generará automáticamente el código en el interior, así como el miembro asociado. Para retomar el ejemplo de arriba, con la propiedad automática, tendríamos el código siguiente:

```
public int AnoCompra { get; private set; }
```

La ganancia de productividad es inmediata y consigue un resultado completamente idéntico. En cambio, hay que constatar que, cuando falta el cuerpo, no es posible personalizar la transformación con las propiedades automáticas.

Aquí puede ver algunas pequeñas aclaraciones adicionales sobre las propiedades:

- El ámbito global debe ser más permisivo que el ámbito de get o de set. Por ejemplo, no es posible tener una propiedad con un ámbito limitado por la palabra clave protected y al mismo tiempo hacerlo accesible en modo lectura de manera pública.
- El uso de get y set no es obligatorio. Se puede tener una propiedad con solo un get (eso significa que está en modo de solo lectura, es decir, que solo se la puede

asignar a la instanciación de la clase). La lógica es la misma con un set, pero su interés es escaso.

- Desde C# 9, hay una palabra clave nueva que puede sustituir a set: init. Esta última significa que la propiedad se puede definir únicamente en la instanciación de la clase, pero de manera más flexible que en ausencia de set. Abordaremos este concepto en la sección Instanciar una clase.
- Desde C# 6, es posible asignar el valor de una propiedad después de su declaración (siempre que esta propiedad sea accesible en modo de escritura). Para hacerlo, solo hay que colocar la asignación después de la declaración, como en el bloque siguiente:

```
public int AnoCompra { get; set; } = 2020;
```

Es posible mostrar un descriptor de acceso un poco especial: una propiedad indexada. Esto permite acceder a un valor de la clase como si esta fuera una colección indexada (estudiaremos las colecciones en el capítulo Algoritmia). Para hacerlo, hay que definir cuál será el tipo de dato que permitirá acceder a una información de la clase.

Por ejemplo, podemos imaginar una clase que describe un garaje, donde se puede acceder a cada coche que contiene usando el número de plaza dentro del garaje. En este caso, el tipo del valor acceso es un int porque se trata de un número entero.

La sintaxis para declarar este descriptor de acceso es un poco específica: se indica el tipo de retorno, seguido por la palabra clave this con corchetes. Dentro de estos últimos, se describe la declaración de la variable de acceso (con su tipo y su nombre).

Para ilustrar esto con el ejemplo anterior, tendríamos la siguiente declaración:

```
public Coche this[int numeroPlaza]
{
    get { ... }
}
```

### 3. Instanciar una clase

Ahora que sabemos cómo definir una clase, la siguiente etapa consiste en descubrir cómo instanciarla.

#### a. El constructor

Antes de ver cómo proceder, hay que comprender el principio detrás de la instanciación. Todas las clases contienen lo que se llama un constructor. Se trata de un método particular, al que se llama de manera automática y obligatoria cuando se instancia una clase. La sintaxis del constructor también es distinta de la de los métodos clásicos; un constructor no puede tener tipo de retorno y su nombre debe corresponder al nombre de la clase:



## ÁMBITO NOMBRE\_DE\_LA\_CLASE(PARÁMETROS)

Por ejemplo, la sintaxis de declaración del constructor de nuestra clase Ordenador sería:

```
public Ordenador() { }
```

El constructor declarado aquí arriba en realidad es muy poco útil. En efecto, el compilador C#, en ausencia de una definición cualquiera de un constructor dentro de la clase, en todos los casos habría generado un garante para estas características: public y sin parámetros.

Es útil definir un constructor en el caso de que deseemos tener un ámbito de método distinto, o incluso poder transmitirle parámetros.

Cuando se ha definido un constructor particular mediante el desarrollador, el compilador C# no genera un constructor public sin parámetro porque eso podría alterar el código. De hecho, sería una pena haber definido un constructor private, y luego que el compilador añadiera un constructor public que modificaría el comportamiento inicialmente deseado por el desarrollador.

Por ejemplo, imaginemos que queremos crear un ordenador informando directamente de su año de compra (comprender: es imposible crear un ordenador sin especificar este valor):

```
public Ordenador(int anoCompra)
{
    AnoCompra = anoCompra;
}
```

En caso de herencia de otra clase, se llama al constructor de la clase de base antes de llamar al constructor de la clase hija. Por ejemplo, si tenemos las siguientes clases:

```
public class Ordenador
{
    public Ordenador()
    {
        Console.WriteLine("Construcción Ordenador");
    }
}
public class Mac : Ordenador
{
    public Mac ()
    {
        Console.WriteLine("Construcción Mac");
    }
}
```

durante la instanciación de la clase Mac, la consola muestra:

## Construcción Ordenador

### Construcción Mac

Si el constructor de la clase padre necesita un parámetro, es posible pasárselo añadiendo, después de la definición del constructor de la clase hija, la llamada al constructor padre usando la palabra clave `base`:

```
public class Ordenador
{
    public Ordenador(int anoCompra)
    {
        Console.WriteLine("Construcción Ordenador");
    }
}
public class Mac : Ordenador
{
    public Mac() : base(2020)
    {
        Console.WriteLine("Construcción Mac");
    }
}
```

Esta llamada es obligatoria si el constructor padre tiene al menos un parámetro. La lógica predeterminada es que el constructor de una clase hija toma al menos los mismos parámetros que el padre, posiblemente más.

Teóricamente, no hay límite para la cantidad de parámetros que puede aceptar un constructor. Sin embargo, hay que permanecer atentos a esta cuestión porque cada una de las futuras instanciaciones necesitará completarlos. Por eso se recomienda no sobrepasar los cinco parámetros para un constructor.

Ahora que se ha definido un constructor, nos falta ver cómo llamar a este último de manera efectiva para instanciar una clase.

### **b. Instanciación con la palabra clave new**

Para crear una instancia nueva de una clase en memoria, se ha reservado una palabra clave: `new`. Gracias a esta última, es posible llamar al constructor de la clase. Así, para crear una instancia nueva de nuestra clase `Ordenador` y guardarla en una variable, la sintaxis que se debe usar es la siguiente:

```
Ordenador orde = new Ordenador(2020);
```

En el código de arriba, hemos creado una variable llamada `orde`, que contiene una instancia nueva de la clase `Ordenador`, y hemos pasado el valor `2020` como parámetro de constructor

para el año de compra. Si el constructor no aceptara parámetros, la llamada sería la siguiente:

```
Mac mac = new Mac();
```

En los dos ejemplos de arriba, es posible sustituir el tipo de la variable por la palabra clave `var` para ir más rápido. El código es lo bastante explícito para que el lector del código dude del tipo de la variable. En cuanto al compilador, este sustituirá la palabra clave `var` por el tipo explícito porque pese a todo C# sigue siendo un lenguaje fuertemente tipado.

Desde la versión 9 del lenguaje C#, ya no es necesario volver a especificar el tipo después de `new`. La nueva sintaxis es más concisa, pero necesita que se especifique el tipo a nivel de la variable:

```
Ordenador orde = new(2020);  
Mac mac = new();
```

Así, no es posible escribir:

```
var orde = new();
```

porque el compilador no sabrá cuál es el tipo que desea la instalación. Por eso hay que elegir entre la nueva sintaxis `new` o el uso de la palabra clave `var`.

De la misma manera, C# ofrece una posibilidad de escritura para asignar directamente las propiedades que se pueden definir después de la creación. Esta sintaxis un poco particular se llama `object initializer`. Consideramos la siguiente clase:

```
public class Persona  
{  
    public string Nombre { get; set; }  
    public string Apellido { get; set; }  
    public bool EsVIP { get; private set; }  
}
```

En la clase que aparece arriba, una `Persona` contiene un nombre y un apellido accesibles de manera pública en modo de lectura y de escritura, y un booleano que indica si es VIP. El booleano solo se puede definir dentro de la clase. Por eso, si queremos usar `object initializer`, no podremos asignar el valor booleano, solo el nombre y el apellido:

```
var persona = new Persona { Nombre = "Christophe",  
    Apellido = "Mommer" };
```

Dado que `set` es accesible de manera pública en modo de escritura, es posible modificar estos valores después de la creación (haciendo `persona.Apellido = "Lolo"`). Aquí vuelve a intervenir la palabra clave introducida por C# 9 `init`, que permite al constructor o a la sintaxis `object initializer` limitar la asignación de la propiedad a la creación.

Debido a la herencia, se puede almacenar un tipo más específico dentro de un tipo más genérico. Eso es posible en la instanciación, pero también en las transferencias de parámetros. Vamos a interesarnos por este concepto, llamado polimorfismo.

## 4. Polimorfismo

La palabra polimorfismo procede de una concatenación de dos términos: poli (significa múltiple) y morfo (significa forma). Esto significa que un objeto puede tomar varias formas.

Concretamente, haciendo un vínculo de herencia, se declara que «si Y hereda de X, entonces Y es un X». Retomando nuestro ejemplo, Mac hereda de Ordenador, entonces un Mac es un Ordenador.

Esto también quiere decir que se puede usar un Mac en todas las posiciones donde se espera un Ordenador. El Mac se «rebajará» al nivel de un Ordenador (no veremos sus características específicas, sino solo lo que tiene en común con el resto de los ordenadores).

Y esto también es válido para las variables:

```
Ordenador mac = new Mac();
```

Esta escritura solo es posible porque Mac hereda de la clase Ordenador. Sin embargo, la operación inversa no es posible. Así, el siguiente código no se compilará:

```
Mac mac = new Ordenador();
```

Por supuesto, el concepto de polimorfismo también se puede aplicar a las variables y a los parámetros. Se puede imaginar una clase nueva Persona que posee un método EscribirUnLibroEnCSharp, que requiere una instancia de Ordenador para funcionar:

```
public class Persona
{
    public void EscribirUnLibroEnCSharp(Ordenador orde)
    {
    }
}
```

Así, se puede crear esta Persona nueva y pasarle cualquier Ordenador:

```
var christophe = new Persona();
christophe.EscribirUnLibroEnCSharp(new Ordenador(2020));
christophe.EscribirUnLibroEnCSharp(new Mac());
```

Este código es completamente funcional en los dos casos. En compensación, a semejanza de las variables, cuando se espera un tipo muy específico, es necesario proporcionar. Así, el siguiente código no funciona:

```
public class Persona
```

```
{  
    public void CrearUnaAplicaciónIPhone(Mac mac) { }  
}  
var christophe = new Persona();  
christophe.CrearUnaAplicaciónIPhone(new Mac());  
christophe.CrearUnaAplicaciónIPhone(new Ordenador()); // aquí el  
compilador presentará un error
```

Sin embargo, es imposible llamar a este método con una instancia de la clase Ordenador porque se espera un tipo más específico.

# Conceptos avanzados

Después de haber visto los conceptos básicos de la POO, es el momento de descubrir algunos conceptos avanzados.

## 1. Herencia avanzada

Cuando hemos abordado la cuestión de la herencia, lo hemos hecho a través del prisma de una clase Y que hereda de otra clase X; estas dos clases se pueden crear.

### a. Métodos virtuales

Cuando una clase hereda de otra, esta disfruta de manera natural de los métodos y de los datos presentes en la clase madre. Sin embargo, para algunas necesidades puede suceder que la clase hija tenga que redefinir algunos métodos. Para que eso sea posible, la clase madre debe declarar estos métodos como virtuales, lo que significa que definen un funcionamiento concreto, pero sigue siendo posible sobrecargar este funcionamiento. En este funcionamiento intervienen dos palabras clave:

- A nivel de la clase madre, es necesario añadir la palabra clave virtual antes del nombre del método.
- A nivel de la clase hija, es necesario añadir la palabra clave override antes del nombre del método que se quiere sustituir.

En concreto, imaginemos que nuestra clase Ordenador dispone de un método Encender, que muestra «Encendido en proceso...», pero se quiere mostrar «Su Mac se inicia...» dentro del caso de la clase Mac. Hay varias maneras de responder a esta necesidad; una posible solución es sustituir el contenido del método Encender:

```
public class Ordenador  
{  
    public virtual void Encender()  
    {
```

```

        Console.WriteLine("Encendido en proceso...");
    }
}

public class Mac : Ordenador
{
    public override void Encender()
    {
        Console.WriteLine("Su Mac se inicia...");
    }
}

```

Cuando se llama al método Encender de un objeto que resulta ser un Mac, la consola muestra «Su Mac se inicia...», y lo mismo en el caso donde se ha usado el concepto de polimorfismo:

```

Ordenador mac = new Mac();
mac.Encender();

```

Esto se debe sencillamente al hecho de que la variable es de tipo Ordenador, pero se ha colocado una instancia de Mac dentro. Durante la llamada, se invoca el método del tipo más preciso. Gracias a eso, tiene la posibilidad de usar un método que trabaja con un tipo genérico, llamando a un método común. Así, cada implementación puede conocer sus propias particularidades sin que eso altere la ejecución.

Sin embargo, con este planteamiento, es necesario que la clase madre declare estos métodos como virtuales y que la clase hija esté informada de ello para que pueda usar la palabra clave override. Otra manera de acercarse a esta especificación es usar una clase abstracta.

## **b. Clase abstracta**

En POO, se puede definir una clase de manera abstracta, es decir, que no puede ser instanciada (por eso es imposible llamar a un new arriba). El principio de esta clase es mutualizar propiedades y métodos para que todas las clases hijas puedan usarlas. De la misma manera, solo se puede definir un método abstracto dentro de una clase abstracta, de modo que cada clase que herede de ella esté obligada a redefinirla.

Para crear una clase abstracta, solo hay que usar la palabra clave abstract en el nivel de la definición:

```

public abstract class Ordenador { }

```

Al usar esta declaración, es imposible escribir el siguiente código (eso hará que aparezca un error de compilación):

```

var ordenador = new Ordenador();

```

Sencillamente porque la clase Ordenador es abstracta y solo sirve de definición básica para las clases que heredan de ella. También es posible definir un método abstracto, usando la misma palabra clave: `abstract`. Un método calificado como abstracto no puede contener cuerpo y forzosamente debe ser redefinido durante una herencia, al contrario que un método virtual. Por ejemplo:

```
public abstract class Ordenador
{
    public string Marca { get; set; }
    public abstract void Encender();
}
public class Mac : Ordenador
{
    public override void Encender()
    {
    }
}
```

Como consecuencia de heredar de una clase abstracta que contiene un método abstracto, es obligatorio redefinir el método abstracto. No redefinirlo provocará un error de compilación. Cabe señalar que es posible tener varios métodos abstractos; en este caso, hay que redefinirlos todos ellos.

Sin embargo, queda una limitación, ya sea en el caso del uso de una clase abstracta o de métodos virtuales: solo es posible heredar de una y solo una clase. Esta limitación se puede eludir multiplicando el contenido dentro de la clase madre (o su propia jerarquía), pero eso no puede ser suficiente ni ser lo ideal. Afortunadamente, hay un concepto que permite a las clases compartir una misma filosofía y hacerlo desde varios elementos: las interfaces.

### **c. Interfaz**

Una interfaz es el equivalente a un contrato. A diferencia de los dos enfoques anteriores, una interfaz no contiene ningún código concreto, sino sólo las definiciones que debe concretar obligatoriamente una clase que implementa esta interfaz. Así, para una clase dada, se pueden implementar varias interfaces y, por lo tanto, respetar varios contratos.

La convención, en C#, quiere que el nombre de una interfaz empiece por una *i* mayúscula, lo que permite distinguirla de una clase.

Para la definición solo hay que usar la palabra reservada `interface`. Dentro de su cuerpo, solo puede contener firmas de métodos o de las declaraciones de propiedades. El concepto de ámbito no existe dentro de una interfaz: allí todo es público por convención.

Por ejemplo:

```
public interface IOrdenador
{
    void Encender();
    string Marca { get; set; }
}
```

```
}
```

Una vez más, gracias al planteamiento polimórfico, una clase concreta se puede pasar a métodos que esperan una interfaz:

```
public class Mac : IOrdenador
{
    public string Marca { get; set; }
    public void Encender() { }
}
public class CentralOrdenadores
{
    public void IniciarOrdenadorPrincipal(IOrdenador ordenador)
    {
        ordenador.Encender();
    }
}
var central = new CentralOrdenadores();
var mac = new Mac();
central.IniciarOrdenadorPrincipal(mac);
```

La herencia sigue siendo posible entre interfaces. Haciendo esta operación, una interfaz agrega ciertas especificaciones a la interfaz de la que hereda. Eso permite respetar un principio de programación S.O.L.I.D. (I = Interface Segregation Principle). Por ejemplo:

```
public interface ITieneUnTeclado
{
    void Escribir(string phrase);
}

public interface IOrdenador : ITieneUnTeclado
{
    void Encender();
}
```

En el código de arriba, la interfaz IOrdenador dispone de un método Encender propio, pero también tiene un método Escribir porque implementa ITieneUnTeclado. Si una clase implementa la interfaz IOrdenador, debe definir los dos métodos debido a la jerarquía de interfaz.

```
public class Mac : IOrdenador
{
    public void Encender() { Console.WriteLine("Su Mac se inicia..."); }
    public void Escribir(string phrase) { Console.WriteLine(phrase); }
}
```

Una vez más, el polimorfismo nos ofrece una flexibilidad operativa bastante agradable:

```
public void EscribirUnaNovela(ITieneUnTeclado maquina)
```



```
{  
}  
var mac = new Mac();  
EscribirUnaNovela(mac);
```

#### **d. Implementación predeterminada en una interfaz**

Incluso si una interfaz debe considerarse como que solo contiene firmas de métodos y de propiedades, desde C# 8 es posible escribir una implementación predeterminada.

Con esta posibilidad, puede hacer evolucionar una interfaz que ya se ha usado sin necesidad de modificar las clases que la implementan. Sobre todo, es útil en el caso de que haya expuesto públicamente una interfaz y de que la implementan clases fuera de su control. Modificar una interfaz hace que sea necesario para todas las clases implementar estas modificaciones, lo que puede romper la compilación de ciertos proyectos. Con ese fin, desde C# 8 se puede definir un comportamiento predeterminado. Retomando nuestra interfaz `IOrdenador`, imaginemos que queremos añadir un método `Apagar`, y eso sin romper la compilación:

```
public interface IOrdenador  
{  
    void Encender();  
    void Apagar() { Console.WriteLine("Apagado en proceso..."); }  
}
```

Como vemos aquí arriba, se le puede dar un cuerpo a un método de interfaz haciendo esta implementación predeterminada.

Incluso si eso permite evitar un error de compilación en las clases que operan en esta interfaz, con frecuencia es recomendable no dar implementación predeterminada, para que los usuarios de la interfaz estén al corriente de la existencia de este método nuevo.

#### **e. Enmascaramiento**

Cuando se ha programado un método para ser redefinido, como hemos estudiado en la sección Métodos virtuales, se puede usar la palabra clave `override`. Sin embargo, se puede redefinir un método, incluso cuando no haya sido inicialmente declarado como virtual. Para eso, se usa la técnica del enmascaramiento.

El enmascaramiento consiste en redefinir un método que ya existe en la clase básica, sin usar la palabra clave `override`, pero sobreescribiendo su definición. Para eso, solo hay que crear la misma firma añadiendo la palabra clave `new`. Esta palabra se debe encontrar entre el ámbito y el tipo de retorno del método para realizar esta operación.

Esto solo es útil si el método es completamente idéntico. Si hay una diferencia en la firma (nombre, tipo de retorno o parámetros), el enmascaramiento es inútil.

Vamos a ilustrarlo mediante un ejemplo:

```
public class Ordenador
{
    public void Encender()
    {
        Console.WriteLine("Encendido en curso...");
    }
}
```

```
public class Mac : Ordenador
{
    public new void Encender()
    {
        Console.WriteLine("Su Mac se inicia...");
    }
}
```

La clase Mac hereda de la clase Ordenador. Como el método Encender no es virtual, la clase Mac debe enmascararlo para indicar su propia implementación.

Por eso, si se ejecuta el siguiente código:

```
var ordenador = new Ordenador();
var mac = new Mac();
ordenador.Encender();
mac.Encender();
```

tendremos la siguiente salida:

```
Encendido en curso...
Su Mac se inicia...
```

Si se elimina la palabra clave new de la firma del método Encender de la clase Mac, el compilador generará el siguiente warning:

```
warning    CS0108:  'Mac.Encender()' enmascara el miembro heredado
'Ordenador.Encender()'. Use la palabra clave new si el enmascaramiento es intencionado.
```

## **f. Prohibir la herencia**

Es posible indicar que un método o una clase se consideran como finales, es decir, que no se puede heredar más de ellos. Para hacerlo se usa la palabra clave sealed.

Por ejemplo, si se considera que la clase Mac no puede ser heredada:

```
public class Ordenador
{
}
public sealed class Mac : Ordenador
{
}
```

```
}  
// el siguiente código provocará un error de compilación  
public class MacMini : Mac  
{  
}
```

También se puede definir eso al nivel de un método de datos.

## 2. Los diferentes tipos de objetos

Hasta ahora, hemos abordado la POO a través del prisma de una clase. Aunque eso constituye el elemento central de la programación orientada a objetos, no es el único, ni mucho menos. En esta sección vamos a ver los otros tipos, y las principales diferencias entre estos últimos y una clase. Pero antes de eso, vamos a echar un vistazo a algunas nociones específicas sobre el concepto de clase.

### a. Tipos de referencia

La gestión de la memoria es automática en C# y .NET, gracias a sistemas inteligentes. En efecto, las herramientas operativas que Microsoft pone a disposición contienen un elemento central para la operación correcta de la aplicación: el recolector de basura (o garbage collector en inglés, con frecuencia abreviado como GC).

El único objetivo de esta herramienta es vigilar la ocupación de la memoria viva de la aplicación y limpiar los objetos que ya no se utilizan. Cada vez que se usa la palabra clave `new` en una clase, se crea una instancia nueva. Para poder usar esta instancia, hay que guardarla en algún lugar.

Además, se dice que una clase es un tipo de **referencia**, es decir, que una variable de tipo clase es un puntero hacia un espacio de memoria reservado para guardar los datos y metadatos de la clase. Aquí el objetivo no es hacer un curso sobre la gestión de la memoria en C# y .NET, sino comprender bien que una clase es una referencia hacia una zona reservada. El GC vigilará esta zona y la limpiará cuando ya no se use. Eso evita enfrentarse al fenómeno denominado pérdidas de memoria, donde la memoria de la aplicación aumenta constantemente, por lo general a causa de un olvido de desasignación.

El tipo de referencia también implica un concepto extremadamente importante: el cero. Dado que se trata de un puntero hacia una zona de memoria, una variable de tipo clase puede apuntar de manera efectiva hacia la zona en cuestión o apuntar hacia el cero. Entonces se dice que el valor es igual a `null`, que también es el valor predeterminado de toda variable de tipo clase no inicializada.

Por ejemplo, al declarar una variable sin hacer instanciación con `new`, la variable existe, pero es igual a `null` porque no se ha reservado ninguna zona de memoria:

```
Mac mac;
```

Finalmente, la definición de la igualdad entre dos valores de tipo de referencia solo se hace sobre la base de la comparación de las direcciones de memoria, y para eso su contenido importa poco. Por supuesto, esta igualdad puede ser redefinida para basarse en otros criterios; pero si no hay especificación, se realiza la comparación de las direcciones de memoria.

El valor null está exclusivamente dedicado a los tipos de referencia. Eso nos permite ver el segundo tipo existente en C#: el tipo de **valor**.

## **b. Tipos de valor**

La diferencia fundamental entre un tipo de referencia y un tipo de valor es que este último puede apuntar al cero. Por eso, cuando no hay especificación, se asigna un valor predeterminado por obligación. Igualmente, hay que saber que un tipo de valor no se guarda en memoria en el mismo lugar que un tipo de referencia. La zona de memoria reservada para los tipos de valor se llama pila, mientras que la de los tipos de referencia se llama montón.

El montón puede contener una gran cantidad de datos (la memoria es virtualmente ilimitada), pero el acceso a estos es lento porque este espacio está fragmentado y el GC tiene que reordenar las cosas que se encuentran dentro de él de manera sistemática. La pila, por su parte, tiene un espacio de almacenamiento reducido, pero es extremadamente rápida porque los datos siempre están apilados de manera lógica, lo que significa que su acceso responde a un algoritmo muy eficaz.

Sin saberlo, ya ha usado un tipo de valor con este libro: `int`. En efecto, `int` descansa sobre un tipo de valor que permite guardar un valor numérico. Por eso, es imposible escribir el siguiente código:

```
int valor = null; // error de compilación
```

Si declara un `int` sin asignar un valor, este último toma el valor programado de manera predeterminada. En este caso específico, es 0.

```
int cero; // el valor será 0
```

Como no es posible guardar null en un `int`, eso significa que `int` no descansa sobre una clase, sino sobre otro tipo de datos: una estructura. Incluso aunque el uso que hará probablemente sea más marginal, conocer la existencia de las estructuras es fundamental para comprender bien el funcionamiento de un programa.

Se puede declarar una estructura usando la palabra clave `struct`:

```
public struct MiEstructura
{
}
```

El contenido de una estructura puede ser casi equivalente a una clase (allí se pueden encontrar propiedades, miembros y métodos):

```

public struct Punto
{
    public int X { get; set; }
    public int Y { get; set; }
}
Punto punto = new Punto();
punto.X = 100;
punto.Y = 200;

```

Se puede crear un constructor en una estructura, pero solo desde C# 10 es posible eliminar (o sustituir) el constructor de manera predeterminada (sin parámetro).

Así, el siguiente código provoca un error de compilación fuera de la versión 10 del lenguaje:

```

public struct Punto
{
    public int X { get; set; }
    public int Y { get; set; }
    public Punto() // El compilador indicará un error aquí
    {
    }
}

```

Un constructor de estructura también tiene una limitación: todas sus propiedades deben asignarse en el constructor. No es posible asignar solo una parte de ellas, so pena de tener un error de compilación.

Para retomar nuestro ejemplo, aquí tenemos un constructor completo:

```

public struct Punto
{
    public int X { get; set; }
    public int Y { get; set; }
    public Punto(int x, int y)
    {
        X = x;
        Y = y;
    }
}

```

Con la declaración de arriba, se pueden crear instancias de la estructura Punto de dos maneras:

```

Punto a = new Punto();
a.X = 100;
a.Y = 200;
Punto b = new Punto(150, 150);

```

La gestión de la memoria es un elemento que hay que considerar con el tipo de valor. En efecto, con el tipo de referencia se manipula una referencia, por lo tanto, un puntero hacia una zona de la memoria. Con un tipo de valor se trata de un objeto completo.

Por eso, una asignación de una variable a otra o un paso como parámetro a una función implica una copia completa de los datos. Para ilustrar esta diferencia, he aquí un pequeño fragmento de código explicativo:

```
public class Persona
{
    public string Nombre { get; set; }
}

public void Main()
{
    int i = 42;
    Increment(i);
    Console.WriteLine(i); // Mostrará 42, porque el valor i
se ha copiado durante la llamada
    var p = new Persona { Nombre = "Nombre" };
    Renombrar(p);
    Console.WriteLine(p.Nombre); // Mostrará "Nombre nuevo" porque
la referencia permite manipular la zona de memoria afectada
}
public void Increment(int i) { i = i + 1; }
public void Renombrar(Persona p) { p.Nombre = "Nombre nuevo"; }
```

Por este motivo no es posible que un tipo de valor sea igual a null, porque no es posible copiar null.

Sin embargo, la versión 2 del lenguaje C# ha aportado flexibilidad de escritura para permitir que un elemento de tipo de valor sea null. Enseguida vamos a detallar el uso de los tipos que aceptan valores null.

### 3. Modificadores de clase

Como se ha enunciado en la sección Principios de la programación orientada a objetos - ¿Qué es una clase?, una clase puede tener modificadores que indican que se comporta de una manera distinta. Estos dos modificadores son static y partial.

#### a. El concepto de static

Una clase declarada como static no se puede instanciar, es decir, que no es posible crear una instancia nueva de esta. Una vez declarada una clase como tal, es imposible que contenga datos o funciones denominadas «de instancia». La totalidad de lo que declara

debe ser calificado como `static`. Se usa esta misma palabra clave para transmitir este concepto:

```
static class NOMBRE_DE_LA_CLASE
{
}
```

La palabra clave `static` también se puede incorporar a los métodos y a las propiedades, permitiendo un acceso sin instancia:

```
static class Herramientas
{
    public static int Value { get; set; }
    public static void Metodo() { }
}
```

Para usar la clase, hay que usar su tipo directamente, de la misma manera que si tuviéramos una instancia de esta clase:

```
var valor = Herramientas.Value;
Herramientas.Metodo();
```

Una clase que no está declarada como `static` también puede contener métodos o propiedades `static`. Estas últimas, en cambio, no pueden usar nada de todo lo relacionado con la instancia. Sin embargo, es posible llamar a un método `static` desde un método de instancia:

```
public class MiClase
{
    public static void Metodo()
    {
        MetodoInstancia(); // ilegal
    }
    public void MetodoInstancia()
    {
        Metodo(); // ok
    }
}
```

De manera general, los métodos y tipos `static` no son un planteamiento recomendado. En algunos casos, pueden resultar útiles e incluso necesarios en otros (métodos de extensión, por ejemplo, que estudiaremos en el capítulo Conceptos avanzados, que aborda los temas avanzados).

Aunque la instrucción `using` en el encabezado del archivo se use para importar un espacio de nombres, también es posible utilizarla para importar un tipo estático. Todo ello, para evitar tener que colocar de manera sistemática un prefijo en el acceso a un método o a una propiedad estática mediante el nombre de la clase. Retomando el ejemplo de la clase estática `Herramientas` descrita arriba, podríamos usarla de la siguiente manera:

```

using static IEspacioNombresHaciaLaClaseEstaticaHerramientas.Herramientas;
public class MiClase
{
    public void Metodo()
    {
        var valorHerramienta = Value;
        Metodo();
    }
}

```

Cabe destacar que, para este caso preciso, es necesario usar la palabra clave `static` justo después de `using`, para indicarle al compilador que se trata de la importación de un tipo estático.

## b. El concepto de clase parcial

Al contrario que `static`, el concepto de clase parcial solo se aplica a la clase y su definición, y no a los métodos o propiedades. La presencia de la palabra clave `partial` indica que la definición de la clase está contenida en varios archivos, ubicados en varios lugares (pero en el mismo espacio de nombres). El compilador se encarga de reagrupar los archivos para producir una única definición durante la fase de compilación. Es la única manera de tener dos archivos que definen la misma clase en el mismo nivel jerárquico (ver error encontrado en el capítulo Primer programa).

Por ejemplo:

En el archivo `MiClase.cs`

```

public partial class MiClase
{
    public int Valor { get; set; }
}

```

En el archivo `MiClase2.cs`

```

public partial class MiClase
{
    public void Incremento() { Valor = Valor + 1; }
}

```

Así, si otra parte de código usa la clase `MiClase`, tiene acceso al contenido definido en los dos archivos:

```

var c = new MiClase();
c.Incremento();
Console.WriteLine(c.Valor);

```

De manera general, no se recomienda usar uno mismo este planteamiento. Este último toma su sentido para completar o personalizar clases que se generan automáticamente mediante las herramientas porque no se pueden redefinir.



# Ejercicio

Ahora que ha adquirido las nociones del desarrollo orientado a objetos, es el momento de pasar a la práctica a través de un ejercicio.

## 1. Enunciado

El objetivo de este ejercicio es poder gestionar un garaje. El garaje realiza varias operaciones en los coches:

- Repintar un coche para cambiar el color.
- Reparar el coche.
- Hacer el mantenimiento del coche y actualizar la fecha del último mantenimiento.

El garaje también se ocupa de los camiones, pero solo para repararlos.

A excepción de asignar o de leer posibles propiedades, las acciones no harán más que mostrar en la consola, gracias al método `WriteLine`, la acción en cuestión.

Un coche tiene marca, color, fecha de revisión y un indicador de buen funcionamiento.

El pequeño escenario para implementar es el siguiente:

- Crear un garaje.
- Crear dos coches (un Peugeot azul y un Ferrari rojo) y un camión.
- Reparar el Peugeot y el camión, y hacer mantenimiento en el Ferrari.
- Repintar el Peugeot de color verde.

## 2. Solución

Tras una simple lectura del enunciado existen, como suele suceder, muchas maneras de resolver este ejercicio correctamente. En este caso, el planteamiento es poner en práctica lo que hemos visto en este capítulo.

Al principio, hay que empezar por crear la jerarquía que permite gestionar los tipos Coche y Camion. Estos últimos se pueden reparar; de ahí la conveniencia de poner en común el indicador de reparación. Esto puede hacerse usando una interfaz (pero eso implica que

cada clase haga la implementación explícita) o una clase de base abstracta. Vamos a impulsar esta solución porque en el enunciado nada indica que sea necesario heredar clases adicionales. De la misma manera, el color se puede guardar como cadena de caracteres, pero para evitar tener «Rojo» y «rojo» como colores, la unificación gracias a una enum parece ser una buena idea.

```
public enum Color
{
    Verde,
    Azul,
    Rojo
}
public abstract class Vehiculo
{
    public bool Repara { get; set; }
}
public class Coche : Vehiculo
{
    public string Marca { get; set; }
    public Color Color { get; set; }
    public DateTime FechaMantenimiento { get; set; }
}
public class Camion : Vehiculo
{
}
```

El planteamiento a partir de la clase abstracta de base Vehiculo nos permite disfrutar del polimorfismo cuando creamos nuestra clase Garaje:

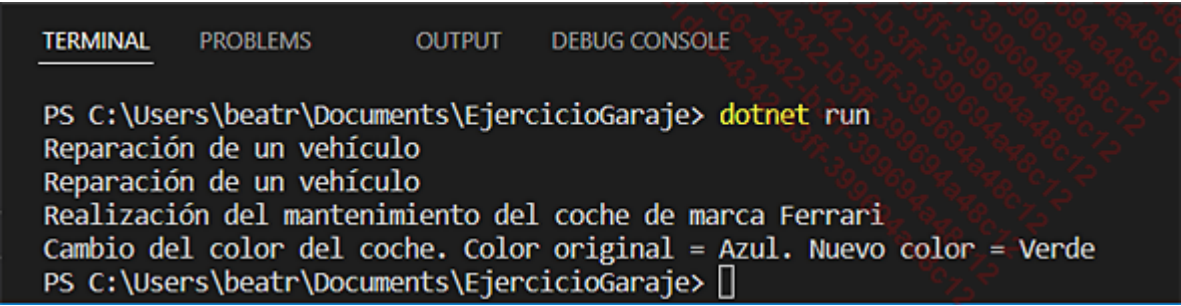
```
public class Garaje
{
    public void HacerMantenimiento(Coche coche)
    {
        Console.WriteLine("Realización del mantenimiento del
coche de marca" + coche.Marca);
        coche.FechaMantenimiento = DateTime.Now;
    }
    public void Repintar(Coche coche, Color nuevoColor)
    {
        Console.WriteLine("Cambio del color del coche.
Color original = " + coche.Color + ". Nuevo color = " +
nuevoColor);
        coche.Color = nuevoColor;
    }
    public void Reparar(Vehiculo vehiculo)
    {
        Console.WriteLine("Reparación de un vehículo");
        vehiculo.Repara = true;
    }
}
```

```
}  
}
```

Ahora, podemos desarrollar nuestro escenario en el archivo Program.cs de nuestra aplicación de consola:

```
var garaje = new Garaje();  
var peugeot = new Coche { Marca = "Peugeot", Color = Color.Azul };  
var ferrari = new Coche { Marca = "Ferrari", Color = Color.Rojo };  
var camion = new Camion();  
garaje.Reparar(camion);  
garaje.Reparar(peugeot);  
garaje.HacerMantenimiento(ferrari);  
garaje.Repintar(peugeot, Color.Verde);
```

La ejecución del programa nos da esto:



```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE  
PS C:\Users\beatr\Documents\EjercicioGaraje> dotnet run  
Reparación de un vehículo  
Reparación de un vehículo  
Realización del mantenimiento del coche de marca Ferrari  
Cambio del color del coche. Color original = Azul. Nuevo color = Verde  
PS C:\Users\beatr\Documents\EjercicioGaraje> █
```