6. Programación orientada a objetos

6.1. ¿Por qué los objetos?

Cuando tenemos que realizar un proyecto grande, será necesario descomponerlo en varios subprogramas, de forma que podamos repartir el trabajo entre varias personas.

Esta descomposición no debe ser arbitraria. Por ejemplo, será deseable que cada bloque tenga unas responsabilidades claras (lo que se conoce como que cada bloque tenga una "alta cohesión"), y que cada bloque no dependa de los detalles internos de otros bloques (lo que se llama "bajo acoplamiento").

Existen varias formas de descomponer un proyecto, pero posiblemente la más recomendable consiste en tratar de verlo como una serie de "objetos" que colaboran entre sí.

Una forma de "descubrir" los objetos que forman parte de un programa es tomar como punto de partida la **descripción** del problema, y **subrayar** los nombres con un color y los verbos con otro color.

Como ejemplo, vamos a dedicar un momento a pensar qué elementos ("objetos") hay en un juego como el clásico **Space Invaders**.

Cuando entramos al juego, aparece una pantalla de bienvenida, que nos recuerda detalles como la cantidad de puntos que obtendremos al "destruir" cada "enemigo":

Y cuando comenzamos una partida, veremos una pantalla como ésta, que vamos a analizar con más detalle:

Observando la pantalla anterior, o (preferiblemente) tras jugar algunas partidas, podríamos hacer una primera descripción del juego en lenguaje natural (que posiblemente habría que refinar más adelante, pero que nos servirá como punto de partida):

Nosotros manejamos una "**nave**", que se puede **mover** a izquierda y derecha y que puede **disparar**. Nuestra nave se esconde detrás de "**torres defensivas**", que se **destruyen** poco a poco cuando les impactan los **disparos**. Nos atacan (nos **disparan**) "**enemigos**". Además, estos enemigos se **mueven** de lado a lado, pero no de forma independiente, sino como un "**bloque**". En concreto, hay tres "tipos" de enemigos, que no se diferencian en su comportamiento, pero sí en su imagen. Tanto nuestro disparo como los de los enemigos **desaparecen** cuando salen de la pantalla o cuando impactan con algo. Si un disparo del enemigo impacta con nosotros, **perderemos una vida**; si un disparo nuestro impacta con un enemigo, lo **destruye**. Además, en ocasiones aparece un "**OVNI**" en la parte superior de la pantalla, en la parte superior de la pantalla, que se **mueve** del lado izquierdo al lado derecho y nos permite obtener puntuación extra si le impactamos con un disparo (y en ese caso, es destruido). Igualmente, hay un "**marcador**", que **muestra** la puntuación actual (que se irá **incrementando**) y el récord (mejor puntuación hasta el momento). La puntuación actual que muestra el marcador se **reinicia** al comienzo de cada partida. Antes de cada "**partida**", pasamos por una pantalla de "**bienvenida**", que muestra una animación que nos informa de cuántos puntos obtenemos al destruir cada tipo de enemigo.

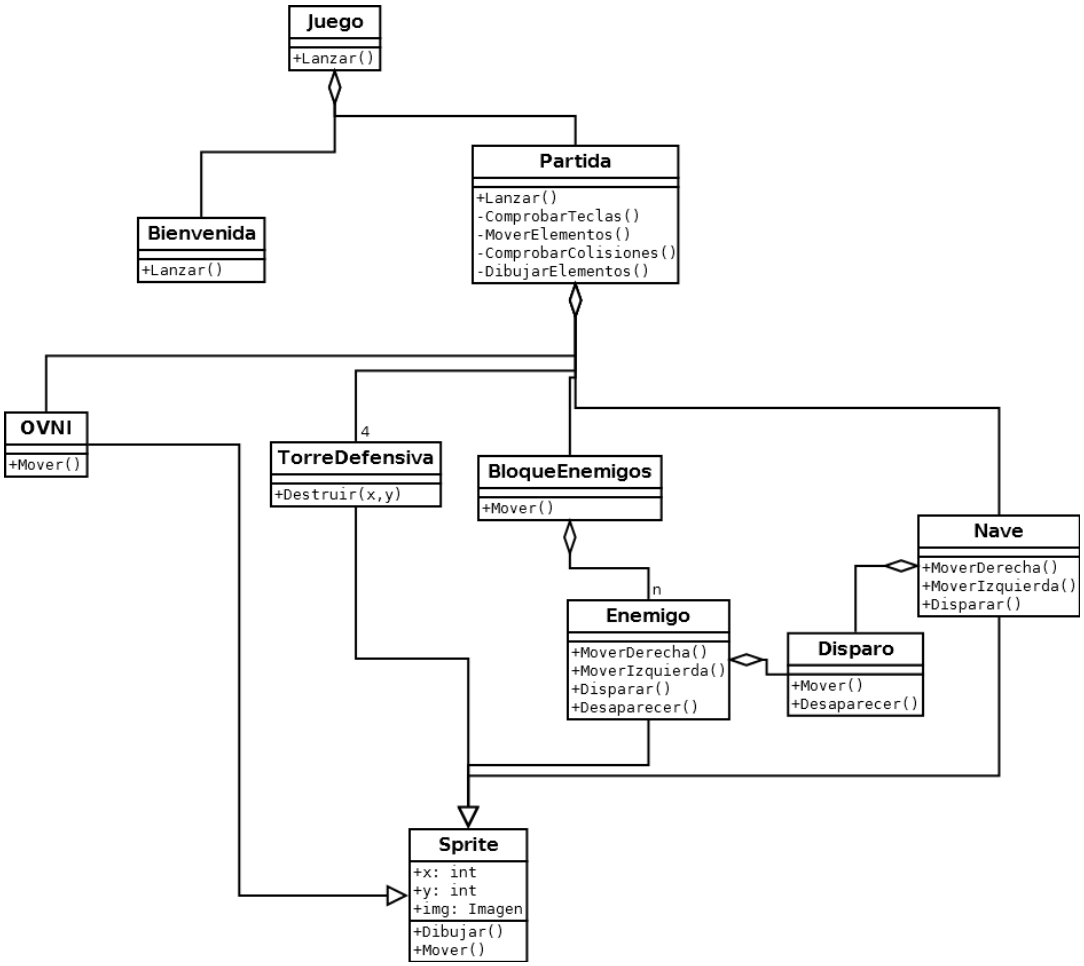
A partir de esa primera descripción, podemos buscar los **nombres** (puede ayudar si los subrayamos con un rotulador de marcar), que indicarán los objetos en los que podemos descomponer el problema, y los **verbos** (con otro rotulador de marcar, en otro color), que indicarán las acciones que puede realizar cada uno de esos objetos.

De la descripción subrayada de este juego concreto podemos extraer los siguientes objetos y las siguientes acciones para cada uno de ellos:

- Nave : mover izquierda, mover derecha, disparar, perder vida
- Torre defensiva: destruir (un fragmento, en ciertas coordenadas)
- Enemigos : mover, disparar, desaparecer
- Ovni : mover, desaparecer
- Bloque (formado por enemigos) : mover
- Disparo : mover, desaparecer
- Marcador : mostrar, reiniciar, incrementar puntos
- Partida (contiene: nave, enemigos, torres, ovni)
- Juego (formado por: bienvenida y partida)
- Bienvenida : lanzar

(En general, esta descomposición no tiene por qué ser única, distintos programadores o analistas pueden llegar a soluciones parcialmente distintas).

Esa serie de objetos, con sus relaciones y sus acciones, se puede expresar mediante un "**diagrama de clases**", que en nuestro caso podría ser así (simplificado):



El nombre "diagrama de clases" se debe a que se llama "**clase**" a un conjunto de objetos que tienen una serie de características comunes. Por ejemplo, un Honda Civic Type-R con matrícula 0001-AAA sería un objeto concreto perteneciente a la clase "Coche". En nuestro juego real existirán varios objetos que pertenezcan a la clase Enemigo, varios de la clase TorreDefensiva, un objeto de la clase Bienvenida, etc.

Algunos de los detalles que se pueden leer de ese diagrama (y que deberían parecerse bastante a la descripción inicial) son:

- El **objeto** principal de nuestro proyecto se llama "Juego" (el diagrama típicamente se leerá de arriba a abajo).
- El juego contiene una "Bienvenida" y una "Partida" (esa relación de que un objeto "**contiene**" a otros se indica mediante un rombo en el extremo de la línea que une ambas clases, junto a la clase "contenedora").
- La "Bienvenida" se puede "Lanzar" al comienzo del juego.
- Si así lo elige el jugador, se puede "Lanzar" una "Partida".
- En una partida participan una "Nave", cuatro "Torres" defensivas, un "BloqueDeEnemigos" formado por varios "Enemigos" y un "Ovni".
- El "Ovni" se puede "Mover".
- Una "TorreDefensiva" se puede "Destruir" poco a poco, a partir de un impacto en ciertas coordenadas x,y.
- El "BloqueDeEnemigos" se puede "Mover".
- Cada "Enemigo" individual se puede mover a la derecha, a la izquierda, puede disparar o puede desaparecer (cuando un disparo le acierte).
- El "Disparo" se puede "Mover" y puede "Desaparecer".
- Nuestra "Nave" se puede mover a la derecha, a la izquierda y puede disparar.
- Tanto la "Nave" como las "Torres", los "Enemigos" y el "Ovni" son subtipos concretos de "Sprite" (esa **relación entre un objeto más genérico y uno más específico** se indica con una línea que termina en punta de flecha, que señala al objeto más genérico).
- La partida también tendrá podrá realizar acciones adicionales, relacionadas con la lógica del juego, como "ComprobarTeclas" (para ver qué teclas ha pulsado el usuario), "MoverElementos" (para actualizar el movimiento de los elementos que deban moverse por ellos mismos, como los enemigos o el OVNI), "ComprobarColisiones" (para ver si dos elementos chocan, como un disparo y un enemigo, y actualizar el estado del juego según corresponda), o "DibujarElementos" (para mostrar en pantalla todos los elementos actualizados).

Y en ese diagrama hemos añadido un detalle avanzado adicional, que nos permitirá introducir ciertos conceptos que luego veremos con más detalle:

- Un "Sprite" es una figura gráfica de las que aparecen en el juego, típicamente un elemento capaz de moverse en pantalla. Cada sprite tendrá detalles como una "imagen" y una posición, dada por sus coordenadas "x" e "y". Además, un sprite será capaz de hacer operaciones como "dibujarse" (aparecer en pantalla) o "moverse" a una nueva posición.

Estos "detalles" de un objeto (como su "x" o su "imagen") los llamaremos "**atributos**". Las "acciones" que un objeto puede realizar (como "mover") las llamaremos "**métodos**". Cuando toda esta estructura de clases que muestra el diagrame se convierta en un programa, los "atributos" se representarán como **variables**, mientras que los "métodos" serán **funciones**.

Los subtipos de sprite (como la "nave" o el "ovni") "**heredarán**" las características de esta clase. Por ejemplo, como un Sprite tiene una coordenada X y una Y, también lo tendrá el OVNI, que es una subclase de Sprite. De igual modo, la nave se podrá "Dibujar" en pantalla, porque también es una subclase de Sprite.

Faltan detalles, pero no es un mal punto de partida. A partir de este diagrama podríamos crear un "esqueleto" de programa que compilase correctamente pero aún "no hiciese nada", y entonces comenzaríamos a repartir trabajo: una persona se podría encargar de crear la pantalla de bienvenida, otra de la lógica del juego, otra del movimiento del bloque de enemigos, otra de las peculiaridades de cada tipo de enemigo, otra del OVNI...

Nosotros no vamos a hacer proyectos tan grandes (al menos, no todavía), pero sí empezaremos a crear proyectos sencillos en los que colaboren varias clases, que permitan sentar las bases para proyectos más complejos, y también ayuden a entender algunas peculiaridades de los temas que veremos a continuación, como el manejo de ficheros en C#.

Como curiosidad, cabe mencionar que en los proyectos grandes es habitual usar **herramientas gráficas** que nos ayuden a visualizar las clases y las relaciones que existen entre ellas, como hemos hecho para el Space Invaders. También se puede dibujar directamente en papel para aclararnos las ideas, pero el empleo de herramientas informáticas tiene varias ventajas adicionales:

- Podemos "arrastrar y soltar" para recolocar las clases, y así conseguir más legibilidad o dejar hueco para nuevas clases que hayamos descubierto que también deberían ser parte del proyecto.

- Algunas de estas herramientas gráficas permiten generar automáticamente un esqueleto del programa, que nosotros rellenaremos después con los detalles de la lógica real.

La metodología más extendida actualmente para diseñar estos objetos y sus interacciones (además de otras muchos detalles de un proyecto que quedan fuera del propósito de este texto) se conoce como **UML** (Unified Modelling Language, lenguaje de modelado unificado). El estándar UML propone distintos tipos de diagramas para representar información, como:

- Los posibles "casos de uso" de una aplicación (posibilidades que la aplicación va a permitir realizar al usuario y "actores" –tipos de usuario- que van a poder realizar cada una de esas actividades).
- Las clases que la van a integrar (que es lo que a nosotros más nos interesa en este momento, y con lo que ya hemos tenido un primer contacto).
- La secuencia de acciones que se debe seguir para realizar las actividades complejas.
- ...

Disponemos de herramientas gratuitas como **ArgoUML**, multiplataforma, que permite crear distintos tipos de diagramas UML y que permite generar el código correspondiente al esqueleto de nuestras clases, o **Dia**, también multiplataforma, pero de uso más genérico (para crear diagramas de cualquier tipo) y que no permite generar código por ella misma pero sí con la ayuda de la herramienta adicional Dia2code.

En los próximos ejemplos partiremos de una única clase en C#, para ampliar posteriormente esa estructura e ir creando proyectos más complejos.

Ejercicio propuesto:
(6.1.1) Piensa en un juego que conozcas, que no sea demasiado complejo (tampoco demasiado simple) y trata de hacer una descripción como la anterior y una descomposición en clases.

6.2. Objetos y clases en C#

Las **clases en C#** se definen de forma parecida a los registros (struct), sólo que ahora, además de variables (que representan sus detalles internos, y que, como ya hemos anticipado, llamaremos sus "**atributos**"), también incluirán funciones (las acciones que puede realizar ese objeto, que llamaremos sus "**métodos**"). Atributos y métodos formarán parte de "un todo", en vez de estar separados en distintas partes del programa. Esta unión de elementos que hasta ahora habríamos considerado algo separado, es lo que se conoce como "**encapsulación**".

Así, una clase "Puerta" se podría declarar así:

```
class Puerta
{

    int ancho;      // Ancho en centímetros
    int alto;       // Alto en centímetros
    int color;      // Color en formato RGB
    bool abierta;   // Abierta o cerrada

    void Abrir()
    {
        abierta = true;
    }

    void Cerrar()
    {
        abierta = false;
    }

    void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }

} // Final de la clase Puerta
```

Como se puede observar, los objetos de la clase "Puerta" tendrán un ancho, un alto, un color, y un estado (abierta o no abierta), y además se podrán abrir o cerrar (también se ha incluido un método para "mostrar su estado", que permite comprobar que todo funciona correctamente).

Un detalle importante que se diferencia de lo que hemos hecho hasta ahora es que, por lo general, cuando estemos hablando de objetos, **las funciones no serán "static"** (salvo Main), por motivos que veremos con detalle dentro de poco.

Para declarar estos objetos que pertenecen a la clase "Puerta", usaremos la palabra "new", igual que hacíamos con los "arrays":

```
    Puerta p = new Puerta();
    p.Abrir();
    p.MostrarEstado();
```

Vamos a completar un **programa de prueba** que use un objeto de esta clase (una "Puerta"), muestre su estado, la abra y vuelva a mostrar su estado:

```
// Ejemplo_06_02a.cs
// Primer ejemplo de clases
```

```
// Introducción a C#, por Nacho Cabanes

using System;

class Puerta
{

    int ancho;     // Ancho en centímetros
    int alto;      // Alto en centímetros
    int color;     // Color en formato RGB
    bool abierta;  // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }

} // Final de la clase Puerta


class Ejemplo_06_02a
{

    static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Vamos a abrir...");
        p.Abrir();
        p.MostrarEstado();
    }
}
```

Este fuente ya no contiene una única clase (class), como todos nuestros ejemplos anteriores, sino dos clases distintas:

- La clase "Puerta", que es el nuevo objeto con el que vamos a practicar. Los detalles (típicamente funciones o "métodos") de esta clase que deban ser utilizados desde otras clases deberán ser "públicos", por lo que deberán

estar precedidos por la palabra "public". Dentro de poco veremos qué otros "especificadores de acceso" podemos emplear, y qué supone cada uno de ellos.

- La clase "Ejemplo_06_02a", que representa a nuestra aplicación. Esta clase no necesita tener nada "público".

(**Nota:** al compilar, puede que obtengas algún "Aviso" -warning- que te dice que has declarado "alto", "ancho" y "color", pero no las estás utilizando; no es importante por ahora, puedes ignorar ese aviso).

El resultado de ese programa es el siguiente:

```
Valores iniciales...
Ancho: 0
Alto: 0
Color: 0
Abierta: False

Vamos a abrir...
Ancho: 0
Alto: 0
Color: 0
Abierta: True
```

Se puede ver que en C# (pero no en todos los lenguajes), las variables que forman parte de una clase (los "atributos") tienen un valor inicial predefinido: 0 para los números, una cadena vacía para las cadenas de texto, "false" para los datos booleanos.

Vemos también que se accede a los métodos y a los datos precediendo el nombre de cada uno por el nombre de la variable y por **un punto**, como hacíamos con los registros (struct).

Aun así, en nuestro caso no podemos hacer directamente "p.abierta = true" desde el programa principal, por dos motivos:

- El atributo "abierta" no tiene delante la palabra "public"; por lo que no es público, sino privado, y no será accesible desde otras clases (en nuestro caso, no lo será desde Ejemplo_06_02a, que es la clase que contiene Main). En el apartado 6.5 veremos cómo dejar claro de forma explícita qué atributos se desea que sean privados, porque otros lenguajes no siguen este convenio.

- Los puristas de la Programación Orientada a Objetos recomiendan que no se acceda directamente a los atributos, sino que siempre se modifiquen usando métodos auxiliares (por ejemplo, nuestro "Abrir"), y que se lea su valor también usando una función. Esto es lo que se conoce como "**ocultación de datos**". Supondrá ventajas como que podremos cambiar los detalles internos de nuestra clase sin que afecte a los programas que puedan utilizarla.

Por ejemplo, para conocer y modificar los valores del "ancho" de una puerta, podríamos crear un método LeerAncho, que nos devolviera su valor, y un método CambiarAncho, que lo reemplazase por otro valor. No hay un convenio claro sobre cómo llamar a estos métodos en español, por lo que es frecuente usar las palabras inglesas "**Get**" y "**Set**" para leer y cambiar un valor, respectivamente. Así, crearemos funciones auxiliares GetXXX y SetXXX que permitan acceder al valor de los atributos (en C# existe una forma alternativa de hacerlo, más compacta, usando "propiedades", que veremos más adelante):

```
int GetAncho()
{
    return ancho;
}

void SetAncho(int nuevoValor)
{
    ancho = nuevoValor;
}
```

(Por no hacer el programa demasiado largo, no crearemos por ahora getters y setters para los demás atributos de una puerta, sólo para el "ancho").

Así, una nueva versión del programa, que incluya ejemplos de Get y Set, podría ser:

```
// Ejemplo_06_02b.cs
// Clases, get y set
// Introducción a C#, por Nacho Cabanes

using System;

class Puerta
{

    int ancho;     // Ancho en centímetros
    int alto;      // Alto en centímetros
    int color;     // Color en formato RGB
    bool abierta;  // Abierta o cerrada
```

```
    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public int GetAncho()
    {
        return ancho;
    }

    public void SetAncho(int nuevoValor)
    {
        ancho = nuevoValor;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }

} // Final de la clase Puerta

class Ejemplo_06_02b
{

    static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Vamos a abrir...");
        p.Abrir();
        p.SetAncho(80);
        p.MostrarEstado();
    }

}
```

Como ya hemos anticipado, puede desconcertar que en "Main" aparezca la palabra "**static**", mientras que no lo hace en los métodos de la clase "Puerta". Veremos el motivo un poco más adelante, pero de momento deberemos perder la costumbre de escribir "static" antes de cada función: si estamos creando objetos, **sólo Main será "static"**.

Ejercicios propuestos:

(6.2.1) Crea una clase llamada Persona, en el fichero "persona.cs". Esta clase deberá tener un atributo "nombre", de tipo string. También deberá tener un método "SetNombre", de tipo void y con un parámetro string, que permita cambiar el valor del nombre. Finalmente, también tendrá un método "Saludar", que escribirá en pantalla "Hola, soy " seguido de su nombre. Crea también una clase llamada PruebaPersona. Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona, les asignará un nombre a cada uno y les pedirá que saluden.

(6.2.2) Tras leer la descripción de Space Invaders que vimos en el apartado anterior, crea una clase Juego, que sólo contenga un método Lanzar, void, sin parámetros, que escriba en pantalla "Bienvenido a Console Invaders. Pulse Intro para salir" y se detendrá hasta que el usuario pulse Intro. Prepara también un Main (en la misma clase), que cree un objeto de la clase Juego y lo lance.

(6.2.3) Para guardar información sobre libros, vamos a comenzar por crear una clase "Libro", que contendrá atributos "autor", "titulo", "ubicacion" (todos ellos strings) y métodos Get y Set adecuados para leer su valor y cambiarlo. Prepara también un Main (en la misma clase), que cree un objeto de la clase Libro, dé valores a sus tres atributos y luego los muestre.

(6.2.4) Crea una clase "Coche", con atributos "marca" (texto), "modelo" (texto), "cilindrada" (número entero), potencia (número real). No hace falta que crees un Main de prueba.

6.3. Proyectos a partir de varios fuentes

En un proyecto grande, es recomendable que **cada clase esté en su propio fichero fuente**, de forma que se puedan localizar con rapidez (no ha sido necesario en los proyectos que hemos hecho hasta ahora, porque eran muy simples).

Es recomendable además (aunque no obligatorio) que cada clase esté en un fichero que tenga el mismo nombre: la clase Puerta debería encontrarse en el fichero "Puerta.cs" (nota: en otros lenguajes, como Java, esto no es una recomendación, sino un requisito).

Para **compilar un programa formado por varios fuentes**, basta con indicar los nombres de todos ellos. Por ejemplo, con Mono sería

```
mcs fuente1.cs fuente2.cs fuente3.cs
```

En ese caso, el ejecutable obtenido tendría como el nombre del primero de los fuentes (fuente1.exe). Podemos cambiar el nombre del ejecutable con la opción "-out" de Mono:

```
mcs fuente1.cs fuente2.cs fuente3.cs -out:ejemplo.exe
```

Si usamos el compilador de línea de comandos incluido en Windows, escribiríamos algo como lo siguiente (en una única línea de consola):

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe fuente1.cs fuente2.cs
fuente3.cs
```

En general, esto no lo podremos hacer de forma sencilla desde editores como Notepad++ y Geany: **Notepad++** tiene un menú "Ejecutar", que nos permite teclear una orden como la anterior, pero no supone ninguna gran ganancia frente a tener abierta una ventana de consola. Como alternativa, **Geany** permite crear "proyectos" formados por varios fuentes, pero no resulta una tarea especialmente cómoda. Por eso, veremos un primer ejemplo de cómo compilar desde "línea de comandos" y luego pasaremos a trabajar con Visual Studio, un entorno mucho más avanzado y con el que ya tuvimos una toma de contacto.

Vamos a **dividir en dos fuentes** el último ejemplo y a ver cómo se compilaría. La primera clase podría ser ésta:

```
// Puerta.cs
// Clases, get y set
// Introducción a C#, por Nacho Cabanes

using System;

class Puerta
{

    int ancho;     // Ancho en centímetros
    int alto;      // Alto en centímetros
    int color;     // Color en formato RGB
    bool abierta;  // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public int GetAncho()
```

```
    {
        return ancho;
    }

    public void SetAncho(int nuevoValor)
    {
        ancho = nuevoValor;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }

} // Final de la clase Puerta
```

Y la segunda clase podría ser:

```
// Ejemplo_06_03a.cs
// Usa la clase Puerta
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_03a
{

    static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Vamos a abrir...");
        p.Abrir();
        p.SetAncho(80);
        p.MostrarEstado();
    }

}
```

Y lo compilaríamos con:

```
mcs ejemplo06_03a.cs puerta.cs -out:ejemploPuerta.exe
```

si empleamos Mono, o bien, con el compilador de línea de comandos de Windows:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe
```

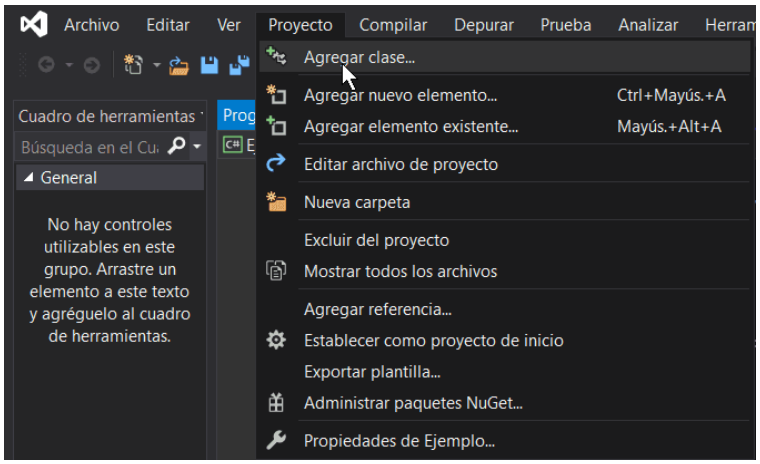
```
-out:ejemploPuerta.exe Ejemplo_06_03a.cs puerta.cs
```

(en una única línea de pantalla) si queremos indicar el nombre del fichero de salida, o, si queremos que se llame como el primer fuente:

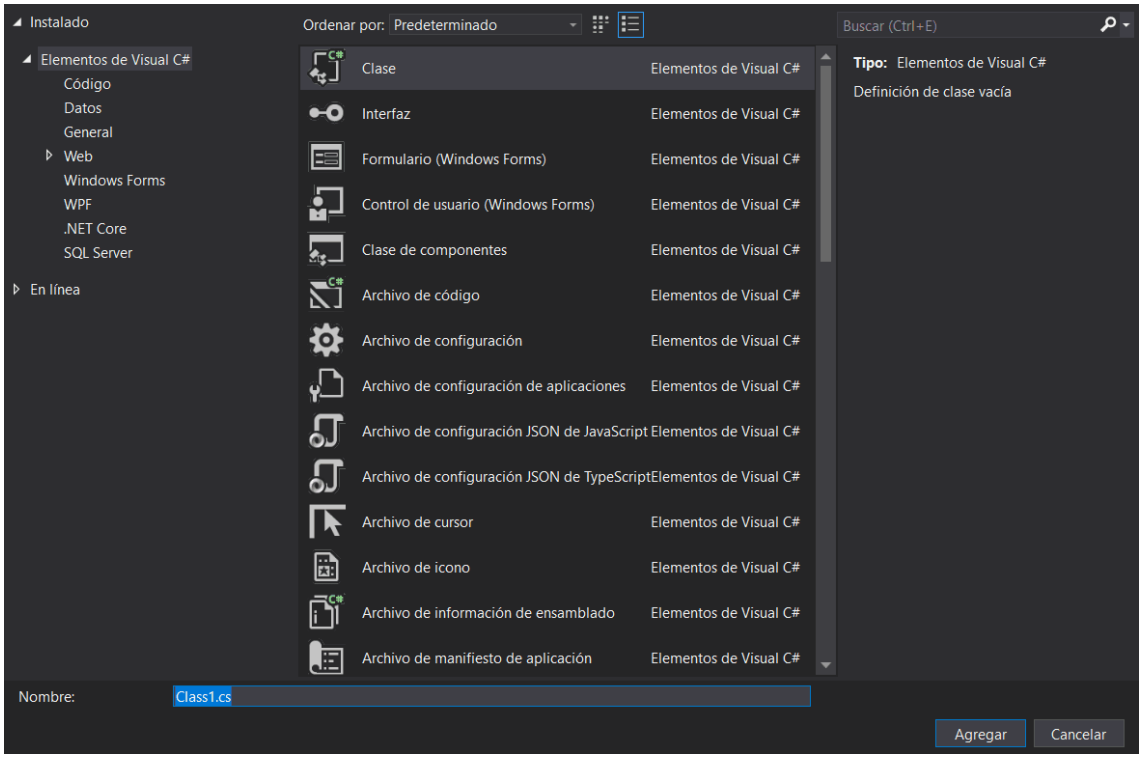
```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe
Ejemplo_06_03a.cs puerta.cs
```

(también en una única línea de pantalla).

Como hemos adelantado, para estos proyectos formados por varias clases, lo ideal es usar algún **entorno más avanzado**, como Visual Studio (que es gratis en su versión Community Edition). Su manejo básico ya lo has visto en el apartado 1.4.3. Cuando se trata de varios fuentes, la única diferencia es que, tras crear el primer fuente, se deberá acudir al menú "Proyecto" para Agregar una nueva clase:



Dentro de los muchos tipos de elementos que se podrían añadir a un proyecto, el que nos interesa es uno llamado simplemente "Clase":



Si la versión más reciente de Visual Studio es demasiado pesada para tu ordenador y resulta lenta, puedes probar una versión más antigua, que encontrarás en el apartado de "older downloads" de la página web de Visual Studio:

https://visualstudio.microsoft.com/es/vs/older-downloads/

También puedes buscar otras alternativas aún más ligeras y también gratuitas, como **SharpDevelop**, disponible para Windows.

En Mac, Visual Studio está basado en lo que antes fue Xamarin Studio (y antes aún MonoDevelop) y es un poco más ligero que la versión de Windows. En Linux, quizá puedas instalar MonoDevelop desde el gestor de paquetes de tu sistema.

Ejercicio propuesto:

(6.3.1) Crea un proyecto con las clases Puerta y Ejemplo_06_03a. Comprueba que todo funciona correctamente.

(6.3.2) Crea un proyecto a partir de la clase Persona (ejercicio 6.2.1), para dividirlo en dos ficheros: la clase Persona estará en el fichero "persona.cs". Una clase PruebaPersona se encontrará en el fichero "pruebaPersona.cs", y deberá contener sólo la función Main, que creará dos objetos de tipo Persona, les asignará un nombre y les pedirá que saluden.

(6.3.3) Crea un proyecto a partir de la clase Libro (ejercicio 6.2.3). El "Main" pasará a una segunda clase llamada "PruebaDeLibro" y desaparecerá de la clase Libro.
(6.3.4) Amplía el esqueleto del ConsoleInvaders (ejercicio 6.2.2): crea un proyecto para Visual Studio o SharpDevelop. Además de la clase "Juego", crea una clase "Bienvenida" y una clase "Partida". El método "Lanzar" de la clase Juego ya no escribirá nada en pantalla, sino que creará un objeto de la clase "Bienvenida" y lo lanzará y luego creará un objeto de la clase "Partida" y lo lanzará. El método Lanzar de la clase Bienvenida escribirá en pantalla "Bienvenido a Console Invaders. Pulse Intro para jugar". El método Lanzar de la clase Partida escribirá en pantalla "Ésta sería la pantalla de juego. Pulse Intro para salir" y se detendrá hasta que el usuario pulse Intro.
(6.3.5) Amplía el esqueleto del ConsoleInvaders (ejercicio 6.3.4): El método Lanzar de la clase Bienvenida escribirá en pantalla "Bienvenido a Console Invaders. Pulse Intro para jugar o ESC para salir". Como veremos con detalle más adelante, puedes comprobar si se pulsa ESC con "`ConsoleKeyInfo tecla = Console.ReadKey(); if (tecla.Key == ConsoleKey.Escape) salir = true;`". El código de la tecla Intro es "ConsoleKey.Enter". También puedes usar "`Console.Clear();`" para borrar la pantalla. Añade un método "GetSalir" a la clase Bienvenida, que devuelva "true" si el usuario ha escogido Salir o "false" si ha elegido Jugar. El método Lanzar de la clase Juego repetirá la secuencia Bienvenida-Partida hasta que el usuario escoja Salir.
(6.3.6) Amplía el esqueleto del ConsoleInvaders (ejercicio 6.3.5): Crea una clase Nave, con atributos "x" e "y" (números enteros, "x" de 0 a 1023 e "y" entre 0 y 767, pensando en una pantalla de 1024x768), e imagen (un string formado por dos caracteres, como "/\"). También tendrá un método MoverA(nuevaX, nuevaY) que lo mueva a una nueva posición, y un método Dibujar, que muestre esa imagen en pantalla (como esta versión es para consola, tendrás que dividir X para que tenga un valor entre 0 y 79, y la Y entre 0 y 23; por ejemplo, puedes dividir la X entre 12 y la Y entre 30). Puedes usar `Console.SetCursorPosition(x,y)` para situarte en unas coordenadas de pantalla. Crea también clase Enemigo, con los mismos atributos. Su imagen podría ser "][". El método Lanzar de la clase Partida creará una nave en las coordenadas (500, 600) y la dibujará, creará un enemigo en las coordenadas (100, 80) y lo dibujará, y finalmente esperará a que el usuario pulse Intro para terminar la falsa sesión de juego.
(6.3.7) Crea un proyecto a partir de la clase Coche (ejercicio 6.2.4): además de la clase Coche, existirá una clase PruebaDeCoche, que contendrá la función "Main", que creará un objeto de tipo coche, pedirá al usuario su marca, modelo, cilindrada y potencia, y luego mostrará en pantalla el valor de esos datos.

6.4. La herencia

Vamos a ver ahora cómo definir una nueva clase de objetos a partir de otra ya existente. Por ejemplo, vamos a crear una clase "Porton" a partir de la clase "Puerta". Un portón tendrá las mismas características que una puerta (ancho, alto, color, abierto o no), pero además se podrá bloquear, lo que supondrá un nuevo atributo y nuevos métodos para bloquear y desbloquear:

```
// Porton.cs
// Clase que hereda de Puerta
// Introducción a C#, por Nacho Cabanes

using System;

class Porton : Puerta
{

    bool bloqueada;

    public void Bloquear()
    {
        bloqueada = true;
    }

    public void Desbloquear()
    {
        bloqueada = false;
    }

}
```

Con "`class Porton: Puerta`" indicamos que Porton debe "heredar" todo lo que ya habíamos definido para Puerta. Por eso, no hace falta indicar nuevamente que un Portón tendrá un cierto ancho, o un color, o que se puede abrir: todo eso lo tiene por ser un "descendiente" de Puerta.

Existen varias **nomenclaturas** para las clases de las que se hereda y las que heredan de ella: superclase y subclase, clase base y clase derivada, clase padre y clase hija...

En este ejemplo, la clase Puerta no cambiaría nada.

No tenemos por qué heredar todo tal y como era; también podemos "redefinir" algo que ya existía. Por ejemplo, nos puede interesar que "MostrarEstado" ahora nos diga también si la puerta está bloqueada. Para eso, basta con volverlo a declarar y añadir la palabra "**new**" para indicar al compilador de C# que sabemos que ya existe ese método y que sabemos seguro que lo queremos redefinir (si no

incluimos la palabra "new", el compilador mostrará un "warning", pero dará el programa como válido; más adelante veremos que existe una alternativa a "new" y en qué momento será adecuado usar cada una de ellas).

```
    public new void MostrarEstado()
    {
        Console.WriteLine("Portón.");
        Console.WriteLine("Bloqueada: {0}", bloqueada);
    }
```

Puedes observar que ese "MostrarEstado" no dice nada sobre el ancho ni el alto del portón. En el próximo apartado veremos cómo acceder a esos datos.

Un programa de prueba, que ampliase el anterior para incluir un "portón", podría ser:

```
// Ejemplo_06_04a.cs
// Portón, que hereda de Puerta
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_04a
{

    static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Vamos a abrir...");
        p.Abrir();
        p.SetAncho(80);
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Ahora el portón...");
        Porton p2 = new Porton();
        p2.SetAncho(300);
        p2.Bloquear();
        p2.MostrarEstado();
    }

}
```

Y su resultado sería:

```
Valores iniciales...
```

```
Ancho: 0
Alto: 0
Color: 0
Abierta: False

Vamos a abrir...
Ancho: 80
Alto: 0
Color: 0
Abierta: True

Ahora el portón...
Portón.
Bloqueada: True
```

Verás que, hasta ahora, ninguno de nuestros "class" empieza con la palabra "**public**", sólo los métodos están etiquetados de esa forma. Dentro de poco hablaremos más sobre los distintos niveles de visibilidad. De momento, basta con que tengas presente que ninguna de tus clases deberán ser públicas, o bien todas deberán serlo. El programa **no compilaría** si la clase Porton fuera pública y la clase Puerta no lo fuera.

Ejercicios propuestos:

(6.4.1) Crea un proyecto con las clases Puerta, Portón y Ejemplo_06_04a. Prueba que todo funciona correctamente.

(6.4.2) Crea una variante ampliada del ejercicio 6.3.2. En ella, la clase Persona no cambia. Se creará una nueva clase PersonaInglesa, en el fichero "personaInglesa.cs". Esta clase deberá heredar las características de la clase "Persona", y añadir un método "TomarTe", de tipo void, que escribirá en pantalla "Estoy tomando té". Crea también una clase llamada PruebaPersona2, en el fichero "pruebaPersona2.cs". Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona y uno de tipo PersonaInglesa, les asignará un nombre, les pedirá que saluden y pedirá a la persona inglesa que tome té.

(6.4.3) Amplía el proyecto del ejercicio 6.3.3 (Libro): crea una clase "Documento", de la que Libro heredará todos sus atributos y métodos. Ahora la clase Libro contendrá sólo un atributo "paginas", número entero, con sus correspondientes Get y Set.

(6.4.4) Amplía el esqueleto del ConsoleInvaders (ejercicio 6.3.6): Crea una clase "Sprite", de la que heredarán "Nave" y "Enemigo". La nueva clase contendrá todos los atributos y métodos que son comunes a las antiguas (todos los existentes, por ahora). A cambio, verás que tanto la nave como el enemigo tendrán la misma “imagen”, pero eso lo solucionaremos pronto.

(6.4.5) Amplía el proyecto de la clase Coche (ejercicio 6.3.7): Crea una clase "Vehiculo", de la que heredarán "Coche" y una nueva clase "Moto". La clase

Vehiculo contendrá todos los atributos y métodos que antes estaban en Coche, y tanto Coche como Moto heredarán de ella.

6.5. Visibilidad

Nuestro ejemplo todavía es poco versátil. Los **métodos** de una "**Puerta**", como "*Abrir()*" y "*Cerrar()*", son públicos, por lo que serán accesibles desde otras clases, pero, por el contrario, los **atributos** de una "Puerta", como el "ancho", el "alto" y el "color", no son accesibles desde ninguna otra clase, ni siquiera desde la clase Porton, porque no tienen ningún indicador de acceso, lo que en C# equivale a decir que son privados (lo que se podría haber detallado de forma explícita con la palabra "**private**").

Nos podríamos sentir tentados de declarar como "public" esos atributos, pero esa no es la solución más razonable, porque no es deseable que sean accesibles desde cualquier sitio, debemos recordar la máxima de "**ocultación de detalles**", que hace nuestros programas sean más fáciles de mantener.

Pero sí sería deseable que esos datos estuvieran disponibles para todos los tipos de Puerta, incluyendo sus "clases hijas", como un Porton. Esto se puede conseguir usando otro modo de acceso: "**protected**". Todo lo que declaremos como "protected" será accesible por las clases derivadas de la actual, pero por ninguna otra:

```
class Puerta
{

    protected int ancho;     // Ancho en centímetros
    protected int alto;      // Alto en centímetros
    protected int color;     // Color en formato RGB
    protected bool abierta;  // Abierta o cerrada

    public void Abrir()
    ...
```

Como ya hemos adelantado, si quisiéramos dejar claro que algún elemento de una clase debe ser totalmente privado (es decir, no accesible ni siquiera por las clases derivadas de la actual), podemos usar la palabra "**private**", en vez de "public" o "protected". En general, será preferible usar "private" en vez de no escribir nada, por legibilidad, para ayudar a detectar errores con mayor facilidad y como costumbre por si más adelante programamos en otros lenguajes, porque puede

ocurrir que en otros lenguajes se considere público (en vez de privado) un atributo cuya visibilidad no hayamos indicado.

Así, un único fuente completo que declarase la clase Puerta, la clase Porton a partir de ella, y que además contuviese un pequeño "Main" de prueba podría ser:

```
// Ejemplo_06_05a.cs
// Portón, que hereda de Puerta, con "protected"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_05a
{

    static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Vamos a abrir...");
        p.Abrir();
        p.SetAncho(80);
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Ahora el portón...");
        Porton p2 = new Porton();
        p2.SetAncho(300);
        p2.Bloquear();
        p2.MostrarEstado();
    }

}

// ----------------------------------------

// Puerta.cs
// Clases, get y set
// Introducción a C#, por Nacho Cabanes

class Puerta
{

    protected int ancho;     // Ancho en centímetros
    protected int alto;      // Alto en centímetros
    protected int color;     // Color en formato RGB
    protected bool abierta;  // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
```

```
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public int GetAncho()
    {
        return ancho;
    }

    public void SetAncho(int nuevoValor)
    {
        ancho = nuevoValor;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }

} // Final de la clase Puerta


// ----------------------------------------

// Porton.cs
// Clase que hereda de Puerta
// Introducción a C#, por Nacho Cabanes

class Porton : Puerta
{

    bool bloqueada;

    public void Bloquear()
    {
        bloqueada = true;
    }

    public void Desbloquear()
    {
        bloqueada = false;
    }

    public new void MostrarEstado()
    {
        Console.WriteLine("Portón.");
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
        Console.WriteLine("Bloqueada: {0}", bloqueada);
    }

}
```

Ejercicios propuestos:

(6.5.1) Crea un proyecto a partir del ejemplo 06.05a, en el que cada clase esté en un fichero separado. Como podrás comprobar, ahora necesitarás un "using System" en cada fuente que vaya a utilizar la Consola.

(6.5.2) Amplía las clases del ejercicio 6.4.2, creando un nuevo proyecto con las siguientes características: La clase Persona no cambia; la clase PersonaInglesa se modificará para que redefina el método "Saludar", para que escriba en pantalla "Hi, I am " seguido de su nombre; se creará una nueva clase PersonaItaliana, en el fichero "personaItaliana.cs", que deberá heredar las características de la clase "Persona", pero redefinir el método "Saludar", para que escriba en pantalla "Ciao"; crea también una clase llamada PruebaPersona3, en el fichero "pruebaPersona3.cs", que deberá contener sólo la función Main y creará un objeto de tipo Persona, uno de tipo PersonaInglesa, uno de tipo PersonaItaliana, les asignará un nombre, les pedirá que saluden y pedirá a la persona inglesa que tome té.

(6.5.3) Retoca el proyecto del ejercicio 6.4.3 (Libro): los atributos de la clase Documento y de la clase Libro serán "protegidos".

(6.5.4) Amplía el esqueleto del ConsoleInvaders (ejercicio 6.4.4): Amplía la clase Nave con un método "MoverDerecha", que aumente su X en 10 unidades, y un "MoverIzquierda", que disminuya su X en 10 unidades. Necesitarás hacer que esos atributos sean "protected" en la clase Sprite. El método "Lanzar" de la clase Partida no esperará hasta el usuario pulse Intro sin hacer nada, sino que ahora usará un do-while que compruebe si pulsa ESC (para salir) o flecha izquierda o flecha derecha (para mover la nave: sus códigos son ConsoleKey.LeftArrow y ConsoleKey. RightArrow). Si se pulsan las flechas, la nave se moverá a un lado o a otro (con los métodos que acabas de crear). Al principio de cada pasada del do-while se borrará la pantalla ("Console.Clear();").

(6.5.5) Mejora el proyecto de la clase Coche (ejercicio 6.4.5): todos los atributos serán "protegidos" y los métodos serán "públicos".

6.6. Constructores y destructores

Hemos visto que, al declarar una clase, automáticamente se dan valores por defecto para los atributos. Por ejemplo, para un número entero, se le da el valor 0. Pero puede ocurrir que nosotros deseemos dar valores iniciales que no sean cero. Esto se puede conseguir creando "setters" para todos los campos y llamándolos uno por uno, o, mejor, declarando un "**constructor**" para la clase.

Un **constructor** es una función especial, que se pone en marcha automáticamente cuando se crea un objeto de una clase, y se suele emplear para dar esos valores iniciales, para reservar memoria si fuera necesario, para leer información desde fichero, etc.

Un constructor se declara usando el mismo nombre que el de la clase, indicador de acceso "public" y sin ningún tipo de retorno. Por ejemplo, un "constructor" para la clase Puerta que le diera los valores iniciales de 100 para el ancho, 200 para el alto, etc., podría ser así:

```
public Puerta()
{
    ancho = 100;
    alto = 200;
    color = 0xFFFFFF;
    abierta = false;
}
```

Podemos tener más de un constructor, cada uno con distintos parámetros. Por ejemplo, puede haber otro constructor que nos permita indicar el ancho y el alto:

```
public Puerta(int an, int al)
{
    ancho = an;
    alto = al;
    color = 0xFFFFFF;
    abierta = false;
}
```

Ahora, si declaramos un objeto de la clase puerta con "Puerta p = new Puerta();", tendrá de ancho 100 y de alto 200, mientras que si lo declaramos con "Puerta p2 = new Puerta(90,220);", tendrá 90 como ancho y 220 como alto.

Un programa de ejemplo que utilizara estos dos constructores para crear dos puertas con características iniciales distintas podría ser:

```
// Ejemplo_06_06a.cs
// Constructores
// Introducción a C#, por Nacho Cabanes

using System;


class Ejemplo_06_06a
{
```

```
    static void Main()
    {
        Puerta p = new Puerta();
        Puerta p2 = new Puerta(90,220);

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();
        Console.WriteLine("Vamos a abrir...");
        p.Abrir();
        p.MostrarEstado();

        Console.WriteLine();
        Console.WriteLine("Para la segunda puerta...");
        p2.MostrarEstado();
    }

}

// ----------------------------------------

class Puerta
{

    protected int ancho;     // Ancho en centímetros
    protected int alto;      // Alto en centímetros
    protected int color;     // Color en formato RGB
    protected bool abierta;  // Abierta o cerrada

    public Puerta()
    {
        ancho = 100;
        alto = 200;
        color = 0xFFFFFF;
        abierta = false;
    }

    public Puerta(int an, int al)
    {
        ancho = an;
        alto = al;
        color = 0xFFFFFF;
        abierta = false;
    }

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
```

```
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }

} // Final de la clase Puerta
```

Al igual que existen los "constructores", también podemos crear un "**destructor**" para una clase, que se encargue de liberar la memoria que pudiéramos haber reservado en nuestra clase (no es nuestro caso, porque aún no sabemos manejar memoria dinámica) o para cerrar ficheros abiertos (que tampoco hemos utilizado). Nuestros programas son tan sencillos que todavía no los necesitarán.

Un destructor se creará con el mismo **nombre** que la clase y que el constructor, pero precedido por el símbolo "~", y no tiene tipo de retorno, ni parámetros, ni especificador de acceso ("public" ni ningún otro), como ocurre en este ejemplo:

```
~Puerta()
{
    // Liberar memoria
    // Cerrar ficheros
}
```

Ejercicios propuestos:

(6.6.1) Amplia las clases del ejercicio 6.5.2, para que todas ellas contengan constructores. Los constructores de casi todas las clases estarán vacíos, excepto del de PersonaInglesa, que prefijará su nombre a "John". Crea también un constructor alternativo para esta clase que permita escoger cualquier otro nombre.

(6.6.2) Amplía el proyecto del ejercicio 6.5.3 (Libro): la clase Libro tendrá un constructor que permita dar valores al autor, el título y la ubicación.

(6.6.3) Amplía el esqueleto del ConsoleInvaders (ejercicio 6.5.4): La clase Nave tendrá un constructor, sin parámetros, que prefijará su posición inicial en las coordenadas 500,600. El constructor de la clase Enemigo recibirá como parámetros las coordenadas X e Y iniciales, para que se puedan cambiar desde el cuerpo del programa. La imagen de la nave y del enemigo se prefijarán en estos constructores.

(6.6.4) Mejora el proyecto de la clase Coche (ejercicio 6.5.5): añade un atributo "cantidadDeRuedas" a la clase Vehiculo, junto con sus Get y Set. El constructor de la clase Coche le dará el valor 4 y el constructor de la clase Moto le dará el valor 2.

6.7. Polimorfismo y sobrecarga

El concepto de "**polimorfismo**" se refiere a que una misma función (un método) puede tener varias formas, ya sea porque reciba distintos tipos de parámetros y/o en distinta cantidad, o incluso porque se aplique a distintos objetos.

Existen dos tipos especialmente importantes de polimorfismo:

- En nuestro último ejemplo (06_06a), los dos constructores "Puerta()" y "Puerta(int ancho, int alto)", se llaman igual pero reciben distintos parámetros, y se comportan de forma que puede ser distinta. Esos constructores son ejemplos de "**sobrecarga**" (también conocida como "polimorfismo ad-hoc"). Es un tipo de polimorfismo en el que el compilador sabe en tiempo de compilación a qué método se debe llamar.

- El caso opuesto es el "**polimorfismo puro**", en el que un mismo método se aplica a distintos objetos de una misma jerarquía (como el "MostrarEstado" del ejemplo 06_05a, que se puede aplicar a una puerta o a un portón), y en ese caso el compilador puede llegar a no ser capaz de saber en tiempo de compilación a qué método se debe llamar, y lo tiene que descubrir en tiempo de ejecución. Es algo que nos encontraremos un poco más adelante.

Ejercicios propuestos:

(6.7.1) A partir de las clases del ejercicio 6.6.1, añade a la clase "Persona" un nuevo método Saludar, que reciba un parámetro, que será el texto que debe decir esa persona cuando salude.

(6.7.2) Amplía el proyecto del ejercicio 6.6.2 (Libro): la clase Libro tendrá un segundo constructor que permita dar valores al autor y el título, pero no a la ubicación, que tomará el valor por defecto "No detallada".

(6.7.3) Amplía el esqueleto del ConsoleInvaders (6.6.3): La clase Enemigo tendrá un segundo constructor, sin parámetros, que prefijará su posición inicial a (100,80) para estas primeras pruebas. La clase Nave tendrá un segundo constructor, con parámetros X e Y, para poder colocar la nave en otra posición desde Main. Verás que hay código repetitivo en esos dos contructores, pero más adelante lo optimizaremos.

(6.7.4) Crea dos nuevos métodos en la clase Vehiculo (ejercicio 6.6.4): uno llamado Circular, que fijará su "velocidad" (un nuevo atributo) a 50, y otro Circular(v), que fijará su velocidad al valor que se indique como parámetro.

6.8. Orden de llamada de los constructores

Cuando creamos objetos de una clase derivada, antes de llamar a su constructor se llama a los constructores de las clases base, empezando por la más general y terminando por la más específica. Por ejemplo, si creamos una clase "GatoSiamés", que deriva de una clase "Gato", que a su vez procede de una clase "Animal", el orden de ejecución de los constructores sería: Animal, Gato, GatoSiames, como se ve en este ejemplo:

```
// Ejemplo_06_08a.cs
// Orden de llamada a los constructores
// Introducción a C#, por Nacho Cabanes

using System;


class Ejemplo_06_08a
{
    static void Main()
    {
        Animal a1     = new Animal();
        Console.WriteLine();

        GatoSiames a2 = new GatoSiames();
        Console.WriteLine();

        Perro a3      = new Perro();
        Console.WriteLine();

        Gato a4       = new Gato();
        Console.WriteLine();
    }
}

// ------------------


class Animal
{
    public Animal()
    {
        Console.WriteLine("Ha nacido un animal");
    }
}

// ------------------

class Perro: Animal
{
    public Perro()
    {
        Console.WriteLine("Ha nacido un perro");
    }
}

// ------------------
```

```
class Gato: Animal
{
    public Gato()
    {
        Console.WriteLine("Ha nacido un gato");
    }
}

// ------------------

class GatoSiames: Gato
{
    public GatoSiames()
    {
        Console.WriteLine("Ha nacido un gato siamés");
    }
}
```

El resultado de este programa es:

```
Ha nacido un animal

Ha nacido un animal
Ha nacido un gato
Ha nacido un gato siamés

Ha nacido un animal
Ha nacido un perro

Ha nacido un animal
Ha nacido un gato
```

Ejercicios propuestos:

(6.8.1) Crea un único fuente que contenga las siguientes clases:

- Una clase Trabajador, cuyo constructor escriba en pantalla "Soy un trabajador".
- Una clase Programador, que derive de Trabajador, cuyo constructor escriba en pantalla "Soy programador".
- Una clase Analista, que derive de Trabajador, cuyo constructor escriba en pantalla "Soy analista".
- Una clase Ingeniero, que derive de Trabajador, cuyo constructor escriba en pantalla "Soy ingeniero".
- Una clase IngenieroInformatico, que derive de Ingeniero, cuyo constructor escriba en pantalla "Soy ingeniero informático".
- Una clase "PruebaDeTrabajadores", que cree un objeto perteneciente a cada una de esas clases.

(6.8.2) Crea una variante del proyecto Libro (ejercicio 6.7.2) en la que el constructor de Documento escriba en pantalla "Creando documento" y el

constructor de Libro escriba en pantalla "Creando libro". Comprueba su funcionamiento.

(6.8.3) Crea una versión alternativa del esqueleto del ConsoleInvaders (6.7.3) en la que el constructor de Sprite escriba en pantalla "Creando sprite" y los constructores de Nave escriba en pantalla "Creando nave en posición prefijada" o "Creando nave en posición indicada por el usuario", según el caso (deberás hacer una pausa para poder verlo antes de que se borre la pantalla). Comprueba su funcionamiento.

(6.8.4) Crea una versión alternativa de las clases Vehiculo, Coche, Moto (6.7.4), que te avise del momento en que se entra a cada constructor. Crea un programa de prueba que defina un coche y una moto, y comprueba su funcionamiento.

6.9. Propiedades

Hasta ahora estábamos siguiendo la política de que los atributos de una clase sean privados, y se acceda a ellos a través de métodos "get" (para leer su valor) y "set" (para cambiarlo). En el caso de C#, existe una forma alternativa de conseguir el mismo efecto, empleando las llamadas "propiedades", que permiten una forma abreviada de escribir sus métodos "get" y "set". Vamos a ver un ejemplo que compare ambas formas de acceder a los datos:

```
// Ejemplo_06_09a.cs
// Ejemplo de propiedades (1)
// Introducción a C#, por Nacho Cabanes

using System;

class EjemploPropiedades
{
    // -------------------------

    // Un atributo convencional, privado
    private int altura = 0;

    // Para ocultar detalles, leemos su valor con un "get"
    public int GetAltura()
    {
        return altura;
    }

    // Y lo fijamos con un "set"
    public  void SetAltura(int nuevoValor)
    {
        altura = nuevoValor;
    }


    // -------------------------

    // Otro atributo convencional, privado
```

```
    private int anchura = 0;

    // Oculto mediante una "propiedad"
    public int Anchura
    {
        get
        {
            return anchura;
        }

        set
        {
            anchura = value;
        }
    }


    // -------------------------

    // El "Main" de prueba
    static void Main()
    {
        EjemploPropiedades ejemplo
            = new EjemploPropiedades();

        // Usando getters y setters
        ejemplo.SetAltura(5);
        Console.WriteLine("La altura es {0}",
            ejemplo.GetAltura() );

        // Usando propiedades
        ejemplo.Anchura = 6;
        Console.WriteLine("La anchura es {0}",
            ejemplo.Anchura );
    }
}
```

Ya habíamos empleado "propiedades" anteriormente, sin saberlo, como por ejemplo la longitud ("Length") de una cadena.

Una curiosidad: si una propiedad tiene un "get", pero no un "set", será una propiedad de sólo lectura, no se nos permitirá escribir órdenes como "Anchura = 4", porque el programa no compilaría. De igual modo, se podría crear una propiedad de sólo escritura, definiendo su "set" pero no su "get".

Ejercicios propuestos

(6.9.1) Crea una nueva versión del ejercicio de la clase Persona (6.7.1), en la que el "nombre" sea una propiedad, con sus correspondientes "get" y "set".

En casos sencillos, como éste, en los que una propiedad se use únicamente para ocultar un dato privado, se puede emplear una **notación más compacta**, en la que ese dato privado ni siquiera existe. Eso permite que nuestro programa dé una

"cara visible" que permite ocultar detalles internos, y que podamos cambiar más adelante dichos detalles internos, pero sin la necesidad de crear esos detalles internos (que serían repetitivos) inicialmente:

```
// Ejemplo_06_09b.cs
// Ejemplo de propiedades (2)
// Introducción a C#, por Nacho Cabanes

using System;

class EjemploPropiedades2
{
    // Propiedades con notación abreviada
    public int Anchura { get; set; }
    public int Altura { get; set; }

    // -------------------------

    // El "Main" de prueba
    static void Main()
    {
        EjemploPropiedades2 ejemplo
            = new EjemploPropiedades2();

        // Usa
        ejemplo.Altura = 5;
        Console.WriteLine("La altura es {0}",
            ejemplo.Altura );

        ejemplo.Anchura = 6;
        Console.WriteLine("La anchura es {0}",
            ejemplo.Anchura );
    }
}
```

(6.9.2) Crea una nueva versión del ejercicio de la clase Persona (6.9.1), en la que el "nombre" sea una propiedad, con "get" y "set" abreviados.
(6.9.3) Crea una clase "Caja" con un atributo "capacidad", que será su capacidad en litros. En esta primera versión no tendremos cajas de más de 250 litros, por lo que el atributo será de tipo "byte". Aun así, para permitir futuras ampliaciones, crea una propiedad "Capacidad", de tipo "int", que oculte ese atributo.

6.10. La palabra "static"

Desde un principio, nos hemos encontrado con que "Main" siempre iba acompañado de la palabra "static". Lo mismo ocurría con las funciones que habíamos creado en el tema 5. En cambio, los métodos (funciones) que pertenecen a nuestros objetos no los estamos declarando como "static". Vamos a ver el motivo:

- La palabra "static" delante de un atributo (una variable) de una clase, indica que es una "variable de clase", es decir, que su valor es el mismo para todos los objetos de la clase. Por ejemplo, si hablamos de coches convencionales, podríamos suponer que el atributo "numeroDeRuedas" va a tener el valor 4 para cualquier objeto que pertenezca a esa clase (cualquier coche). Por eso, se podría declarar como "static". En el mundo real, esto de que todos los objetos de una clase tengan exactamente el mismo valor para una propiedad es muy poco habitual, y por eso casi nunca usaremos atributos "static" en programas orientados a objetos (por ejemplo, no todos los coches que podamos encontrar tendrán 4 ruedas, aunque esa sea la cantidad más frecuente, al igual que podemos encontrar animales que no tengan cuatro patas o dos ojos, aunque sean casos excepcionales).

- Por otra parte, si un método (una función) está precedido por la palabra "static", indica que es un "método de clase", es decir, un método que se podría usar sin necesidad de declarar ningún objeto de la clase. Por ejemplo, podemos crear una clase "Hardware" que represente una consola básico, y podemos crear en ella una función "BorrarPantalla" que se pueda utilizar sin necesidad de crear primero un objeto perteneciente a esa clase, así:

```
// Ejemplo_06_10a.cs
// Métodos "static"
// Introducción a C#, por Nacho Cabanes

using System;

class Hardware
{
    public static void BorrarPantalla()
    {
        for (byte i = 0; i < 25; i ++)
            Console.WriteLine();
    }
}
```

```
class Ejemplo_06_10a
{
    static void Main()
    {
        Console.WriteLine("Pulsa Intro para borrar");
        Console.ReadLine();
        Hardware.BorrarPantalla();
        Console.WriteLine("Borrado!");
    }
}
```

Muchas de las funciones que hemos empleado desde nuestros primeros ejercicios eran estáticas. Por ejemplo, el método "**WriteLine**" de la clase "Console" lo es: no necesitamos crear un objeto de la clase Console para acceder a la función WriteLine.

Nota: si nuestro programa estuviera formado sólo por una clase, y dentro de ella estuviera también "Main", como ocurría en nuestra toma de contacto con las **funciones (tema 5)**, ni siquiera haría falta incluir el nombre de la clase antes de llamar al método estático:

```
// Ejemplo_06_10a2.cs
// Métodos "static" y una única clase
// Introducción a C#, por Nacho Cabanes

using System;

class Hardware
{
    public static void BorrarPantalla()
    {
        for (byte i = 0; i < 25; i ++)
            Console.WriteLine();
    }

    static void Main()
    {
        Console.WriteLine("Pulsa Intro para borrar");
        Console.ReadLine();
        BorrarPantalla();   // Misma clase, no hace falta "Hardware."
        Console.WriteLine("Borrado!");
    }
}
```

Desde una función "static" **no se puede** llamar a otras funciones que no lo sean. Por eso, como nuestro "Main" debe ser static (es un requisito del lenguaje), deberemos siempre elegir entre:

- Que todas las demás funciones de nuestro fuente también estén declaradas como "static", por lo que podrán ser utilizadas directamente desde "Main" (como hicimos en el tema 5, cuando empezamos a practicar el manejo de "funciones" pero aún no conocíamos las "clases").
- Declarar un objeto de la clase correspondiente, y entonces sí podremos acceder a sus métodos desde "Main".

Así, una versión alternativa del ejemplo 06_10a, en la que sí se cree un objeto perteneciente a la clase Hardware (por lo que no será necesario emplear "static"), sería:

```
// Ejemplo_06_10b.cs
// Alternativa a 06_10a, sin métodos "static"
// Introducción a C#, por Nacho Cabanes

using System;

class Hardware
{
    public void BorrarPantalla()
    {
        for (byte i = 0; i < 25; i ++)
            Console.WriteLine();
    }
}

class Ejemplo_06_10b
{
    static void Main()
    {
        Console.WriteLine("Pulsa Intro para borrar");
        Console.ReadLine();
        Hardware miPantalla = new Hardware();
        miPantalla.BorrarPantalla();
        Console.WriteLine("Borrado!");
    }
}
```

Ejercicios propuestos:

(6.10.1) Amplía el ejemplo 06_10a con un función "static" llamada "EscribirCentrado", que escriba centrado horizontalmente el texto que se le indique como parámetro.

(6.10.2) Amplía el ejemplo 06_10b con un función llamada "EscribirCentrado", que escriba centrado horizontalmente el texto que se le indique como parámetro. Al contrario que en el ejercicio 6.10.1, esta versión no será "static".

(6.10.3) Crea una nueva versión del ejercicio 5.2.3 (base de datos de ficheros, descompuesta en funciones), en la que los métodos y variables no sean "static".

6.11. Arrays de objetos

Es muy frecuente que no nos baste con tener un único objeto de una clase, sino que necesitemos manipular varios objetos pertenecientes a la misma clase. En ese caso, podemos almacenar todos ellos en un "array".

Deberemos usar "new" **dos veces**: primero para reservar memoria para el array, y luego para cada uno de los elementos. Por ejemplo, podríamos partir del ejemplo del apartado 6.8 y crear un array de 5 perros así:

```
    Perro[] misPerros = new Perro[5];
    for (byte i = 0; i < 5; i ++)
        misPerros[i] = new Perro();
```

Un fuente completo de ejemplo podría ser

```
// Ejemplo_06_11a.cs
// Array de objetos
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_11a
{
    static void Main()
    {
        Perro[] misPerros = new Perro[5];
        for (byte i = 0; i < 5; i ++)
            misPerros[i] = new Perro();
    }
}

// ------------------

class Animal
{
    public Animal()
    {
        Console.WriteLine("Ha nacido un animal");
    }
}

// ------------------

class Perro: Animal
{
    public Perro()
    {
        Console.WriteLine("Ha nacido un perro");
    }
}
```

y su salida en pantalla, que recuerda a la del apartado 6.8, sería

```
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un perro
```

Ejercicio propuesto:

(6.11.1) Crea una versión ampliada del ejercicio 6.8.1 (clase Trabajador y relacionadas), en la que no se cree un único objeto de cada clase, sino un array de tres objetos.

(6.11.2) Amplía el proyecto Libro (ejercicio 6.7.2), de modo que permita guardar hasta 1.000 libros. "Main" mostrará un menú que permita añadir un nuevo libro o ver los datos de los ya existentes.

(6.11.3) Amplía el esqueleto del ConsoleInvaders (6.7.3), para que haya 10 enemigos en una misma fila (todos compartirán una misma coordenada Y, pero tendrán distinta coordenada X). Necesitarás usar el constructor en la clase Enemigo que recibe los parámetros X e Y.

Además, existe una peculiaridad curiosa: podemos crear un array de "Animales", que realmente **contenga objetos de distintas subclases** (en este caso, unos de ellos serán perros, otros serán gatos, etc.),

```
Animal[] misAnimales = new Animal[3];

misAnimales[0] = new Perro();
misAnimales[1] = new Gato();
misAnimales[2] = new GatoSiames();
```

Un ejemplo más detallado:

```
// Ejemplo_06_11b.cs
// Array de objetos de distintas clases
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_11b
{
```

```
    static void Main()
    {
        Animal[] misAnimales = new Animal[8];

        misAnimales[0] = new Perro();
        misAnimales[1] = new Gato();
        misAnimales[2] = new GatoSiames();

        for (byte i=3; i<7; i++)
            misAnimales[i] = new Perro();

        misAnimales[7] = new Animal();
    }
}

// ------------------

class Animal
{
    public Animal()
    {
        Console.WriteLine();
        Console.WriteLine("Ha nacido un animal");
    }
}

// ------------------

class Perro: Animal
{
    public Perro()
    {
        Console.WriteLine("Ha nacido un perro");
    }
}

// ------------------

class Gato: Animal
{
    public Gato()
    {
        Console.WriteLine("Ha nacido un gato");
    }
}


// ------------------

class GatoSiames: Gato
{
    public GatoSiames()
    {
        Console.WriteLine("Ha nacido un gato siamés");
    }
}
```

La salida de este programa sería:

```
Ha nacido un animal
Ha nacido un perro

Ha nacido un animal
Ha nacido un gato

Ha nacido un animal
Ha nacido un gato
Ha nacido un gato siamés

Ha nacido un animal
Ha nacido un perro

Ha nacido un animal
Ha nacido un perro

Ha nacido un animal
Ha nacido un perro

Ha nacido un animal
Ha nacido un perro

Ha nacido un animal
```

Si los objetos tienen otros métodos, no sólo el constructor, quizá no funcione todo correctamente al primer intento. Pronto veremos los posibles problemas y la forma de solucionarlos.

Ejercicios propuestos:

(6.11.4) A partir del ejemplo 06_11b y del ejercicio 6.8.1 (clase Trabajador y relacionadas), crea un array de trabajadores en el que no sean todos de la misma clase.

(6.11.5) Amplía el proyecto Libro (ejercicio 6.11.2), de modo que permita guardar 1000 documentos de cualquier tipo. A la hora de añadir un documento, se preguntará al usuario si desea guardar un documento genérico o un libro, para usar el constructor adecuado.

(6.11.6) Amplía el esqueleto del ConsoleInvaders (6.11.3), para que haya tres tipos de enemigos, y un array que contenga 3x10 enemigos (3 filas, cada una con 10 enemigos de un mismo tipo, pero distinto del tipo de los elementos de las otras filas). Cada tipo de enemigos será una subclase de Enemigo, que se distinguirá por usar una "imagen" diferente. Puedes usar la "imagen" que quieras (siempre que sea un string de letras, como "}{" o "XX"). Si estas imágenes no se muestran correctamente en pantalla al lanzar una partida (que es posible, según el planteamiento que elijas), no te preocupes por ahora, lo solucionaremos en el siguiente apartado.

6.12. Funciones virtuales. La palabra "override"

En el ejemplo anterior hemos visto cómo crear un array de objetos, usando sólo la clase base en el momento de definirlo, pero insertando realmente objetos de cada una de las clases derivadas, y hemos comprobado que los constructores se llaman correctamente... pero con los métodos podemos encontrar problemas.

Vamos a verlo con un ejemplo que, en vez de tener constructores, va a tener un único método "Hablar", que se redefinirá en cada una de las clases hijas, y después comentaremos qué ocurre al ejecutarlo y cómo solucionarlo:

```
// Ejemplo_06_12a.cs
// Array de objetos, sin "virtual"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_12a
{
    static void Main()
    {
        // Primero creamos un animal de cada tipo
        Perro miPerro = new Perro();
        Gato miGato = new Gato();
        Animal miAnimal = new Animal();

        miPerro.Hablar();
        miGato.Hablar();
        miAnimal.Hablar();

        // Linea en blanco, por legibilidad
        Console.WriteLine();

        // Ahora los creamos desde un array
        Animal[] misAnimales = new Animal[3];

        misAnimales[0] = new Perro();
        misAnimales[1] = new Gato();
        misAnimales[2] = new Animal();

        misAnimales[0].Hablar();
        misAnimales[1].Hablar();
        misAnimales[2].Hablar();
    }
}

// ------------------

class Animal
{
    public void Hablar()
    {
        Console.WriteLine("Estoy comunicándome...");
    }
}
```

```
// ------------------

class Perro: Animal
{
    public new void Hablar()
    {
        Console.WriteLine("Guau!");
    }
}

// ------------------

class Gato: Animal
{

    public new void Hablar()
    {
        Console.WriteLine("Miauuu");
    }
}
```

(Recuerda que, como vimos en el apartado 6.4, ese "**new**" que aparece en "new void Hablar" sirve simplemente para anular un "warning" del compilador, que nos decía algo como "estás redefiniendo este método; añade la palabra new para indicar que eso es lo que deseas". Por tanto, añadir esa palabra "new" al comienzo de cada función equivale a un "sí, sé que estoy redefiniendo este método").

La salida de este programa es:

```
Guau!
Miauuu
Estoy comunicándome...

Estoy comunicándome...
Estoy comunicándome...
Estoy comunicándome...
```

La primera parte era de esperar: si creamos un perro, debería decir "Guau", un gato debería decir "Miau" y un animal genérico debería comunicarse. Eso es lo que se consigue con este fragmento:

```
Perro miPerro = new Perro();
Gato miGato = new Gato();
Animal miAnimal = new Animal();

miPerro.Hablar();
miGato.Hablar();
miAnimal.Hablar();
```

En cambio, si creamos un array de animales, no se comporta correctamente, a pesar de que después digamos que el primer elemento del array es un perro, el segundo es un gato y el tercero es un animal genérico:

```
Animal[] misAnimales = new Animal[3];

misAnimales[0] = new Perro();
misAnimales[1] = new Gato();
misAnimales[2] = new Animal();

misAnimales[0].Hablar();
misAnimales[1].Hablar();
misAnimales[2].Hablar();
```

Es decir, como la clase base es "Animal", y el array es de "animales", cada elemento hace lo que corresponde a un Animal genérico (dice "Estoy comunicándome"), a pesar de que hayamos indicado que se trata de un Perro u otra subclase distinta.

Generalmente, no será esto lo que deseemos. Sería interesante que no fuera necesario crear un array de perros y otro de gatos, sino que bastase con crear un único array de animales, pero que **pudiera contener distintos tipos** de animales.

Para conseguir este comportamiento, debemos **indicar** a nuestro compilador que el método "Hablar" que se usa en la clase Animal **quizá sea redefinido** por otras clases hijas, y que, en ese caso, debe prevalecer lo que indiquen las clases hijas.

La forma de conseguirlo es declarar ese método "Hablar" como "**virtual**" (para indicar al compilador que ese método probablemente será redefinido en las subclases), y emplear en las clases hijas la palabra "**override**" en vez de "new" (para dejar claro que debe reemplazar al método "virtual" original), así:

```
// Ejemplo_06_12b.cs
// Array de objetos, con "virtual"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_12b
{
    static void Main()
    {
        // Primero creamos un animal de cada tipo
        Perro miPerro = new Perro();
        Gato miGato = new Gato();
        Animal miAnimal = new Animal();

        miPerro.Hablar();
        miGato.Hablar();
```

```
        miAnimal.Hablar();

        // Linea en blanco, por legibilidad
        Console.WriteLine();

        // Ahora los creamos desde un array
        Animal[] misAnimales = new Animal[3];

        misAnimales[0] = new Perro();
        misAnimales[1] = new Gato();
        misAnimales[2] = new Animal();

        misAnimales[0].Hablar();
        misAnimales[1].Hablar();
        misAnimales[2].Hablar();
    }
}

// ------------------

class Animal
{
    public virtual void Hablar()
    {
        Console.WriteLine("Estoy comunicándome...");
    }
}

// ------------------

class Perro: Animal
{
    public override void Hablar()
    {
        Console.WriteLine("Guau!");
    }
}

// ------------------

class Gato: Animal
{

    public override void Hablar()
    {
        Console.WriteLine("Miauuu");
    }
}
```

El resultado de este programa ya sí es el que posiblemente deseábamos: tenemos un array de animales, pero cada uno "Habla" como corresponde a su especie:

```
Guau!
Miauuu
Estoy comunicándome...

Guau!
```

```
Miauuu
Estoy comunicándome...
```

Nota: Esto que nos acaba de ocurrir va a ser frecuente. Si el compilador nos avisa de que deberíamos añadir la palabra "new" porque estamos redefiniendo algo... es habitual que realmente lo que necesitemos sea "virtual" en la clase base y "override" en las clases derivadas, especialmente si no se trata de un programa trivial, sino que vamos a tener objetos de varias clases coexistiendo en el programa, y especialmente si todos esos objetos van a ser parte de un único array o de alguna estructura similar (como las "listas", que veremos más adelante).

Ejercicios propuestos:

(6.12.1) Crea una versión ampliada del ejercicio 6.5.1 (Persona, PersonaInglesa, etc.), en la que se cree un único array que contenga personas de varios tipos.

(6.12.2) Crea una variante del ejercicio 6.11.4 (array de Trabajador y derivadas), en la que se cree un único array "de trabajadores", que contenga un objeto de cada clase, y exista un método "Saludar" que se redefina en todas las clases hijas, usando "new" y probándolo desde "Main".

(6.12.3) Crea una variante del ejercicio anterior (6.12.2), que use "override" en vez de "new".

(6.12.4) Amplía el proyecto Libro (ejercicio 6.11.5): tanto la clase Documento como la clase Libro, tendrán un método ToString, que devuelva una cadena de texto formada por título, autor y ubicación, separados por guiones. Crea una clase Articulo, que añada el campo "procedencia". El cuerpo del programa permitirá añadir Artículos o Libros, no documentos genéricos. El método ToString deberá mostrar también el número de páginas de un libro y la procedencia de un artículo. La opción de mostrar datos llamará a los correspondientes métodos ToString. Recuerda usar "virtual" y "override" si en un primer momento no se comporta como debe.

(6.12.5) Amplía el esqueleto de ConsoleInvaders (6.11.6) para que muestre las imágenes correctas de los enemigos, usando "virtual" y "override". Además, cada tipo de enemigos debe ser de un color distinto. (Nota: como veremos más adelante, para cambiar colores puedes usar construcciones como Console.ForegroundColor = ConsoleColor.Green;). La nave que maneja el usuario debe ser blanca.

6.13. Llamando a un método de la clase "padre"

Puede ocurrir que en un método de una clase hija no nos interese redefinir por completo las posibilidades del método equivalente de la superclase, sino ampliarlas. En ese caso, no hace falta que volvamos a teclear todo lo que hacía el

método de la clase base, sino que podemos llamarlo directamente, precediéndolo de la palabra "base".

Por ejemplo, podemos hacer que un Gato Siamés hable igual que un Gato normal, pero diciendo "Pfff" después, así:

```
    public new void Hablar()
    {
        base.Hablar();
        Console.WriteLine("Pfff");
    }
```

Este podría ser un fuente completo:

```
// Ejemplo_06_13a.cs
// Ejemplo de clases: Llamar a la superclase
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_13a
{

    static void Main()
    {
        Gato miGato = new Gato();
        GatoSiames miGato2 = new GatoSiames();

        miGato.Hablar();
        Console.WriteLine();  // Linea en blanco
        miGato2.Hablar();
    }

}

// ------------------

class Animal
{
    // Todos los detalles están en las subclases
}

// ------------------

class Gato: Animal
{
    public void Hablar()
    {
        Console.WriteLine("Miauuu");
    }
}

// ------------------

class GatoSiames: Gato
```

```
{
    public new void Hablar()
    {
        base.Hablar();
        Console.WriteLine("Pfff");
    }
}
```

Su resultado sería

```
Miauuu

Miauuu
Pfff
```

También podemos llamar a un **constructor** de la clase base desde un constructor de una clase derivada. Por ejemplo, si tenemos una clase "RectanguloRelleno" que hereda de otra clase "Rectangulo" y queremos que el constructor de "RectanguloRelleno" que recibe las coordenadas "x" e "y" se base en el constructor equivalente de la clase "Rectangulo", lo haríamos así:

```
public RectanguloRelleno (int x, int y )
    : base (x, y)
{
    // Pasos adicionales
    // que no da un rectangulo "normal"
}
```

(Si no hacemos esto, el constructor de RectanguloRelleno se basaría **en el constructor sin parámetros** de Rectangulo, en vez de hacerlo en el que tiene x e y como parámetros).

Ejercicios propuestos:

(6.13.1) Crea una versión ampliada del ejercicio 6.12.3, en la que el método "Hablar" de todas las clases hijas se apoye en el de la clase "Trabajador".

(6.13.2) Crea una versión ampliada del ejercicio 6.13.1, en la que el constructor de todas las clases hijas se apoye en el de la clase "Trabajador".

(6.13.3) Refina el proyecto Libro (ejercicio 6.12.4), para que el método ToString de la clase Libro se apoye en el de la clase Documento, y también lo haga el de la clase Articulo.

(6.13.4) Amplía el esqueleto de ConsoleInvaders (6.12.5) para que en la clase Enemigo haya un único constructor que reciba las coordenadas X e Y iniciales. Los constructores de los tres tipos de enemigos deben basarse en éste.

6.14. La palabra "this": el objeto actual

Podemos hacer referencia al objeto que estamos usando, con "this". Un primer uso es dar valor a los atributos, incluso si los parámetros tienen el mismo nombre que éstos:

```
public void MoverA (int x, int y)
{
    this.x = x;
    this.y = y;
}
```

Un fuente completo sería así:

```
// Ejemplo_06_14a.cs
// Primer ejemplo de uso de "this": parámetros
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_14a
{
    static void Main()
    {
        Titulo t = new Titulo(38,5,"Hola");
        t.Escribir();
    }
}

// ------------------

class Titulo
{
    private int x;
    private int y;
    private string texto;

    public Titulo(int x, int y, string texto)
    {
        this.x = x;
        this.y = y;
        this.texto = texto;
    }

    public void Escribir()
    {
        Console.SetCursorPosition(x,y);
        Console.WriteLine(texto);
    }
}
```

En muchos casos, podemos evitar este uso de "this". Por ejemplo, podemos emplear otro nombre en los parámetros:

```
public void MoverA (int nuevaX, int nuevaY)
{
    this.x = nuevaX;
    this.y = nuevaY;
}
```

Y en ese caso se puede (y se suele) omitir "this":

```
public void MoverA (int nuevaX, int nuevaY)
{
    x = nuevaX;
    y = nuevaY;
}
```

Pero "this" tiene también otros usos. Por ejemplo, podemos crear un **constructor** a partir de otro que tenga distintos parámetros:

```
public RectanguloRelleno (int x)
{
    columna = x;
    fila = 5;
}

public RectanguloRelleno (int x, int y) : this (x)
{
    fila = y;
    // Pasos adicionales, si los hay
}
```

En ese ejemplo, el primer constructor de RectanguloRelleno recibe sólo la coordenada X (y prefija el valor de Y a 5, por poner un ejemplo arbitario), y también existe este otro constructor, que recibe X e Y, y que se basa en el anterior para fijar el valor de X.

También podría ser al contrario, que el constructor que tiene menos parámetros se apoye en el que tiene más parámetros, así:

```
public Rectangulo (int x, int y)
{
    columna = x;
    fila = y;
}

public Rectangulo (int x) : this (x , 10)
{
    // Pasos adicionales, si los hay
}
```

En este caso, si se usa una construcción como "r = new Rectangulo(5)", el resultado sería el mismo que si se hubiera escrito "r = new Rectangulo(5,10)".

Un fuente completo de ejemplo podría ser así:

```
// Ejemplo_06_14b.cs
// Segundo ejemplo de uso de "this": constructores
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_14b
{
    static void Main()
    {
        Titulo t = new Titulo(38,5,"Hola");
        t.Escribir();
    }
}

// ------------------

class Titulo
{
    private int x;
    private int y;
    private string texto;

    public Titulo(int nuevaX, int nuevaY, string txt)
    {
        x = nuevaX;
        y = nuevaY;
        texto = txt;
    }

    public Titulo(int nuevaY, string txt)
        : this (40-txt.Length/2, nuevaY, txt)
    {
    }

    public void Escribir()
    {
        Console.SetCursorPosition(x,y);
        Console.WriteLine(texto);
    }
}
```

La palabra "this" se usa también para que **unos objetos "conozcan" a los otros**. Por ejemplo, en un juego de 2 jugadores, podríamos tener una clase Jugador, hacer que hereden de ella las clases Jugador1 y Jugador2, que serán muy parecidas entre sí, y nos podríamos sentir tentados de hacer que la clase Jugador tenga un "adversario" como atributo, y que el Jugador1 indique que su adversario es de tipo Jugador2, y lo contrario para el otro jugador (lo que simplificará ciertas

tareas, como que se persigan entre ellos). Esto se podría plantear (de forma incorrecta) así:

```
// Ejemplo_06_14c.cs
// Dos clases que se usan mutuamente: incorrecta, recursividad indirecta
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_14c
{
    public static void Main()
    {
        Jugador1 j1 = new Jugador1();
        Jugador2 j2 = new Jugador2();
    }
}

// ------------------

class Jugador
{
    protected Jugador adversario;
}

// ------------------

class Jugador1 : Jugador
{
    public Jugador1()
    {
        adversario = new Jugador2();
    }
}

// ------------------

class Jugador2 : Jugador
{
    public Jugador2()
    {
        adversario = new Jugador1();
    }
}
```

Cuando lanzamos este programa, se queda "bloqueado" un instante, hasta terminar finalmente lanzando una **excepción de desbordamiento de pila** (Stack Overflow), porque estamos creando una recursividad indirecta sin fin: el jugador1 crea un jugador2, éste crea un nuevo jugador1 (en vez de utilizar el ya existente), que a su vez crea un nuevo jugador2 y así sucesivamente, hasta finalmente desbordar la zona de memoria que está reservada para llamadas recursivas.

Una alternativa mucho menos peligrosa (pero que, a cambio, complica el programa principal), es que sea el programa principal el que indique a cada jugador quién es su adversario:

```
// Ejemplo_06_14d.cs
// Dos clases que se usan mutuamente: "Main" coordina
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_14d
{
    static void Main()
    {
        Jugador j1 = new Jugador();
        Jugador j2 = new Jugador();
        j1.SetAdversario(j2);
        j2.SetAdversario(j1);
    }
}

// ------------------

class Jugador
{
    protected Jugador adversario;

    public void SetAdversario(Jugador nuevoAdversario)
    {
        adversario = nuevoAdversario;
    }
}
```

Otra alternativa es que un Jugador le pueda decir a otro "yo soy tu adversario", y ese "yo" equivaldrá a un "this". En general, eso simplificará el programa principal, a cambio de complicar ligeramente las clases auxiliares:

```
// Ejemplo_06_14e.cs
// Dos clases que se usan mutuamente: "this"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_06_14e
{
    static void Main()
    {
        Jugador j1 = new Jugador();   // Creado sin más detalles
        Jugador j2 = new Jugador(j1); // Indicando su adversario
    }
}

// ------------------
```

```
class Jugador
{
    protected Jugador adversario;

    public Jugador()
    {
    }

    public Jugador(Jugador adversario)
    {
        // Mi adversario es el que me indican
        SetAdversario( adversario );

        // Y yo soy su adversario
        adversario.SetAdversario( this );
    }

    public void SetAdversario(Jugador nuevoAdversario)
    {
        adversario = nuevoAdversario;
    }
}
```

Hay que recordar que, en general, **cuando una clase contiene a otras**, la clase contenedora sabe los detalles de las clases contenidas (la "casa" conoce a sus "puertas"), pero las clases contenidas no saben nada de la clase que las contiene (las "puertas" no saben otros detalles de la "casa" a la que pertenecen). Si queremos que se puedan comunicar con la clase contenedora, deberemos usar "this" para que la conozcan, en vez de crear una nueva clase contenedora con "new", o provocaremos una recursividad indirecta sin fin, como en el primer ejemplo.

Ejercicios propuestos:

(6.14.1) Crea una versión ampliada del ejercicio 6.13.2, en la que el constructor sin parámetros de la clase "Trabajador" se apoye en otro constructor que reciba como parámetro el nombre de esa persona. La versión sin parámetros asignará el valor "Nombre no detallado" al nombre de esa persona.

(6.14.2) Crea una clase Puerta con un ancho, un alto y un método "MostrarEstado" que muestre su ancho y su alto. Crea una clase Casa, que contenga 3 puertas y otro método "MostrarEstado" que escriba "Casa" y luego muestre el estado de sus tres puertas.

(6.14.3) Crea una clase Casa, con una superficie (por ejemplo, 90 m^2) y un método "MostrarEstado" que escriba su superficie. Cada casa debe contener 3 puertas. Las puertas tendrán un ancho, un alto y un método "MostrarEstado" que muestre su ancho y su alto y la superficie de la casa en la que se encuentran. Crea un

programa de prueba que cree una casa y muestre sus datos y los de sus tres puertas.

(6.14.4) Amplía el esqueleto de ConsoleInvaders (6.13.4), de modo que el constructor sin parámetros de la clase Nave se apoye en el constructor con parámetros de la misma clase, prefijando las coordenadas que te parezcan más adecuadas.

6.15. Sobrecarga de operadores

Los "operadores" son los símbolos que se emplean para indicar ciertas operaciones. Por ejemplo, el operador "+" se usa para indicar que queremos sumar dos números.

Pues bien, en C# se puede "sobrecargar" operadores, es decir, redefinir su significado, lo que nos permite sumar (por ejemplo) objetos que nosotros hayamos creado, de forma más cómoda y legible. Así, para sumar dos matrices, en vez de hacer algo como "matriz3 = suma(matriz1, matriz2)" podríamos hacer simplemente " matriz3 = matriz1 + matriz2"

No entraremos mucho más en detalle, pero la idea es que redefiniríamos un método **estático** llamado "**operator** +", y que devolvería un dato del mismo tipo que el que estamos manejando (por ejemplo, una Matriz) y recibiría dos datos de ese mismo tipo como parámetros:

```
public static Matriz operator +(Matriz mat1, Matriz mat2)
{
    Matriz nuevaMatriz = new Matriz();

    for (int x=0; x < tamanyo; x++)
        for (int y=0; y < tamanyo; y++)
            nuevaMatriz[x, y] = mat1[x, y] + mat2[x, y];

    return nuevaMatriz;
}
```

Desde "Main", calcularíamos una matriz como suma de otras dos haciendo simplemente

```
Matriz matriz3 = matriz1 + matriz2;
```

Un ejemplo completo, que muestre cómo sumar vectores en el plano, podría ser así:

```
// Ejemplo_06_15a.cs
// Redefiniendo el operador +
// Introducción a C#, por Nacho Cabanes

using System;

class Vector2D
{
    private double x;
    private double y;

    public Vector2D(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public double GetX()
    {
        return x;
    }

    public double GetY()
    {
        return y;
    }

    public void SetX(double newX)
    {
        x = newX;
    }

    public void SetY(double newY)
    {
        y = newY;
    }

    public override string ToString()
    {
        return "<" + x + "," + y  + ">";
    }

    public void Sumar(Vector2D v2)
    {
        x += v2.x;
        y += v2.y;
    }

    public static Vector2D operator + (Vector2D v1, Vector2D v2)
    {
        Vector2D v3 = new Vector2D(v1.x + v2.x, v1.y + v2.y);
        return v3;
    }
}


class Ejemplo_06_15a
{
    static void Main()
    {
```

```
        Vector2D v = new Vector2D(10, -5);
        Console.WriteLine(v);

        Vector2D v2 = new Vector2D(3, 7);
        v.Sumar(v2);
        Console.WriteLine(v);

        Vector2D v3 = new Vector2D(0, 6);
        Vector2D v4 = new Vector2D(-1, 4.5);
        Vector2D v5 = v3 + v4;
        Console.WriteLine(v5);
    }
}
```

Eso sí, debes tener presente que "no todo se puede redefinir". Por ejemplo, puedes pensar en sobrecargar los operadores [y] para acceder a un cierto elemento de la matriz, pero los corchetes están entre los operadores que C# no permite redefinir (éste es un detalle que dependerá de cada lenguaje).

Si deseas poder acceder a una cierta posición, en general deberías crear métodos Get para leer desde una posición y métodos Set para cambiar el valor guardado en ella. En el caso de C#, también existe la posibilidad de usar "indexadores", con una sintaxis distinta a los operadores que hemos visto, y que por lo general será similar a una propiedad llamada "int this [int index]". El siguiente ejemplo amplía la clase Vector2D para permitir leer la primera coordenada con [0] y la segunda con [1]:

```
// Ejemplo_06_15b.cs
// Indexadores para redefinir []
// Introducción a C#, por Nacho Cabanes

using System;

class Vector2D
{
    private double x;
    private double y;

    public Vector2D(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public double GetX()
    {
        return x;
    }

    public double GetY()
    {
        return y;
    }
```

```
    public void SetX(double newX)
    {
        x = newX;
    }

    public void SetY(double newY)
    {
        y = newY;
    }

    public override string ToString()
    {
        return "<" + x + "," + y  + ">";
    }

    public void Sumar(Vector2D v2)
    {
        x += v2.x;
        y += v2.y;
    }

    public static Vector2D operator + (Vector2D v1, Vector2D v2)
    {
        Vector2D v3 = new Vector2D(v1.x + v2.x, v1.y + v2.y);
        return v3;
    }

    public double this[int posicion]
    {
        get
        {
            if (posicion == 0) return x;
            else if (posicion == 1) return y;
            else throw new IndexOutOfRangeException();
        }

        set
        {
            if (posicion == 0) x = value;
            else if (posicion == 1) y = value;
            else throw new IndexOutOfRangeException();
        }
    }
}


class Ejemplo_06_15b
{
    static void Main()
    {
        Vector2D v = new Vector2D(10, -5);
        Console.WriteLine(v);

        Vector2D v2 = new Vector2D(3, 7);
        v2[0] = 1;
        v2[1] = -8;
        Console.WriteLine(v2);
        Console.WriteLine(v2[1]);
    }
```

```
}
```

Ejercicios propuestos:

(6.15.1) Desarrolla una clase "Matriz", que represente a una matriz de 3x3, con métodos para indicar el valor que hay en una posición, leer el valor de una posición, escribir la matriz en pantalla y sumar dos matrices. (crea métodos "get" y "set" para leer los valores de posiciones de la matriz y para cambiar su valor).

(6.15.2) Si has estudiado los "números complejos", crea una clase "Complejo", que represente a un número complejo (formado por una parte "real" y una parte "imaginaria"). Incluye un constructor que reciba ambos datos como parámetros. Crea también métodos "get" y "set" para leer y modificar los valores de dichos datos, así como métodos que permitan saber los valores de su módulo y su argumento. Crea un método que permita sumar un complejo a otro, y redefine el operador "+" como forma alternativa de realizar esa operación.

(6.15.3) Crea una clase "Fraccion", que represente a una fracción, formada por un numerador y un denominador. Crea un constructor que reciba ambos datos como parámetros y otro constructor que reciba sólo el numerador. Crea un método Escribir, que escriba la fracción en la forma "3 / 2". Redefine los operadores "+", "-", "*" y "/" para que permitan realizar las operaciones habituales con fracciones.

(6.15.4) Crea una clase "Vector3", que represente a un vector en el espacio de 3 dimensiones. Redefine los operadores "+" y "-" para sumar y restar dos vectores, "*" para hallar el producto escalar de dos vectores y "%" para calcular su producto vectorial.

6.16. Clases abstractas e interfaces[2]

Existen otros conceptos algo más avanzados y particulares que conviene conocer, ya que están presentes también en otros lenguajes de programación, como Java, y permiten dotar a las aplicaciones de ciertas concreciones o particularidades imposibles de obtener con la herencia "clásica". Es el caso, entre otros, de las clases abstractas, clases selladas y las interfaces.

6.16.1. Clases abstractas

Las clases abstractas se emplean para definir clases incompletas, es decir, con parte de su código pendiente de implementar más adelante. Por lo tanto, parece lógico pensar que no deberíamos poder crear objetos de esas clases, ya que tendrían su funcionalidad incompleta o reducida, y así ocurre.

[2] Apartado elaborado por Nacho Iborra

Una clase abstracta tiene las siguientes propiedades principales:

- No se pueden instanciar, es decir, no podemos crear un objeto directamente a partir de esa clase. Por ejemplo, si la clase *A* es abstracta, no podemos hacer *new A(...)*. Sin embargo, sí pueden tener constructores, que servirán como base para las subclases.
- Pueden tener elementos abstractos (por implementar), pero no es necesario para que sean abstractas.

La utilidad de las clases abstractas radica en el hecho de poder definir una clase base con una parte de código implementada, y dejar a las subclases la obligación de implementar lo que falte.

Veamos un ejemplo concreto para entenderlo mejor. Imaginemos que queremos definir diferentes tipos de animales (perros, gatos, etc), de forma que todos van a tener un nombre de pila, pero queremos que cada uno hable, o emita su sonido característico (por ejemplo, que los perros digan "Guau", o los gatos digan "Miau"). La mejor forma de solucionar esto es definir una clase base abstracta, que podemos llamar *Animal*, donde dejaremos definido lo común para todos los animales (un atributo para el nombre de pila y un constructor para asignarlo), y definiremos un método abstracto *Hablar* para que cada subtipo de animal emita su propio sonido.

Para definir esta clase abstracta, anteponemos el modificador *abstract* a la palabra *class.* También usaremos este término en cada método que queramos que sea abstracto, justo detrás del modificador de acceso. Nuestra clase *Animal* quedaría así:

```
abstract class Animal
{
    string nombre;

    public Animal(string nombre)
    {
        this.nombre = nombre;
    }

    public abstract void Hablar();
}
```

Ahora vamos a definir un tipo concreto de animal, por ejemplo, un perro. Crearemos la clase *Perro* que heredará de *Animal* (con la herencia convencional vista anteriormente) e implementaremos el método que nos falta (*Hablar*):

```
class Perro : Animal
{
    public Perro(string nombre) : base(nombre)
    {
    }

    public override void Hablar()
    {
        Console.WriteLine("Guau!");
    }
}
```

Se puede observar que, para implementar el método abstracto de la clase base, hay que redefinirlo (*override*). De lo contrario obtendremos un error de compilación que indicará que la subclase no implementa el método abstracto heredado.

6.16.1.1. Clases abstractas a partir de otras

A partir de una clase base abstracta, se puede definir otra clase abstracta que herede de ella. Esto permitiría tener diferentes subtipos de una clase base para los que aún no se puede tener una implementación completa.

Volviendo a nuestro ejemplo anterior con la clase *Animal*, podríamos definir un subtipo llamado *Ave*. Este subtipo también es muy genérico, y nos puede interesar que no se puedan crear instancias de él. Por lo tanto, definimos la clase *Ave* como abstracta, y así no tendríamos necesidad de implementar los métodos abstractos de la clase *Animal*. Además, podemos añadir nuevos métodos abstractos propios de ese subtipo, como por ejemplo, un método *Volar*. La subclase quedaría así:

```
abstract class Ave : Animal
{
    public Ave(string nombre) : base (nombre)
    {
    }

    public abstract void Volar();
}
```

Cuando definamos algún subtipo concreto de *Ave* (por ejemplo, una clase *Pato*), entonces sí estaríamos obligados a implementar todos los métodos abstractos que hayamos heredado de las clases superiores:

```
class Pato : Ave
{
    public Pato(string nombre) : base (nombre)
    {
    }
```

```
    public override void Hablar()
    {
        Console.WriteLine("Cuac Cuac!");
    }

    public override void Volar()
    {
        Console.WriteLine("Estoy volando");
    }
}
```

6.16.1.2. Clases abstractas sin métodos abstractos

Una clase abstracta no necesariamente tiene que tener un método abstracto. Podemos definir el contenido de la clase de forma normal (con sus atributos y métodos), y definir la clase como abstracta. Eso significará que hay cosas pendientes de implementar, que deberán concretarse en las subclases.

Por ejemplo, si no quisiéramos "obligar" a que los subtipos de animal debieran hablar, pero tampoco deseáramos que se pudiese crear ningún *Animal* genérico, podríamos hacer nuestra clase *Animal* abstracta, aunque no tuviera nada abstracto dentro:

```
abstract class Animal
{
    string nombre;

    public Animal(string nombre)
    {
        this.nombre = nombre;
    }
}
```

De esta forma, no podremos tener objetos de tipo *Animal*, sino creados a partir de sus subclases concretas, pero no obligamos a que las subclases tengan que implementar ningún método específico.

6.16.2. Interfaces

Una interfaz es un elemento propio de la programación orientada a objetos que permite definir un conjunto de métodos pendientes de implementar. De esta forma, cualquier clase que quiera heredar esa interfaz, debe implementar los métodos que contiene.

La utilidad de este elemento puede no quedar muy clara con esta definición, pero pensemos en un ejemplo práctico. Imaginemos que queremos definir un conjunto de figuras geométricas: cuadrados, círculos, etc, y queremos que para todas ellas se puedan obtener algunos datos, como por ejemplo el área o el perímetro de las

figuras. Si nos queremos asegurar de que para cualquier figura geométrica se pueda calcular su área y su perímetro, tenemos dos opciones:

- Definir una clase padre *FiguraGeometrica*, que debería ser abstracta, con dos métodos abstractos por implementar (*CalcularArea()* y *CalcularPerimetro()*, por ejemplo), y que el resto de figuras hereden de ella. Decimos que debería ser abstracta porque no hay forma de calcular el área o el perímetro de una figura sin saber de qué figura se trata y sus datos (longitud del lado, radio, etc.)
- Definir una interfaz con el esqueleto de los métodos a implementar, y hacer que el resto de figuras hereden (implementen) esa interfaz.

De las dos opciones, es más aconsejable la segunda, ya que la primera nos dejaría una clase vacía, con dos métodos abstractos por implementar, y nos impediría que cualquier subclase de *FiguraGeometrica* pudiera heredar de otra cosa diferente (porque en C# no se permite la herencia múltiple, que sí se puede llegar a imitar utilizando interfaces).

Así pues, para estos casos podemos definir una interfaz. Éstas se definen con la palabra clave *interface* (en lugar de *class*) y poniendo las cabeceras de los métodos a implementar, finalizadas por punto y coma. Es habitual encontrarse que los nombres de las interfaces comiencen por una I mayúscula (por ejemplo, *IFiguraGeometrica*), aunque no es algo obligatorio, sino un convenio que se suele seguir. Nuestra interfaz podría quedar así:

```
interface IFiguraGeometrica
{
    double CalcularArea();
    double CalcularPerimetro();
}
```

Observemos que los métodos de la interfaz no tienen modificador de acceso (*public, private*), ya que eso será definido en las clases que la implementen. Ahora podemos definir una clase que herede de esa interfaz. En realidad, cuando heredamos de una interfaz no estamos heredando realmente, sino **implementando** la interfaz, aunque la forma de expresar en C# que una clase implementa una interfaz es la misma que para la herencia: se define la clase, dos puntos y después el nombre de la interfaz a implementar. Si por ejemplo creamos la clase *Cuadrado* y hacemos que implemente la interfaz anterior, quedaría algo así:

```
class Cuadrado : IFiguraGeometrica
```

```
{
    double lado;

    public Cuadrado(double lado)
    {
        this.lado = lado;
    }

    public double CalcularArea()
    {
        return lado * lado;
    }

    public double CalcularPerimetro()
    {
        return 4 * lado;
    }
}
```

De la misma forma, podemos definir otras clases que implementen la interfaz, como por ejemplo, la clase *Circulo*.

```
class Circulo : IFiguraGeometrica
{
    double radio;

    public Circulo(double radio)
    {
        this.radio = radio;
    }

    public double CalcularArea()
    {
        return Math.PI * radio * radio;
    }

    public double CalcularPerimetro()
    {
        return 2 * Math.PI * radio;
    }
}
```

Además, podemos hacer que cualquier clase implemente más de una interfaz. Así, por ejemplo, si tenemos otra interfaz llamada *IDibujable* con un método *Dibujar*...

```
interface IDibujable
{
    void Dibujar();
}
```

Podríamos hacer que nuestras figuras geométricas, además de serlo, implementen esta otra interfaz para, por ejemplo, dibujarse en pantalla. Para ello, ponemos separadas por comas todas las interfaces que queramos implementar en cada clase:

```
class Cuadrado : IFiguraGeometrica, IDibujable
{
    double lado;

    public Cuadrado(double lado)
    {
        this.lado = lado;
    }

    public double CalcularArea()
    {
        return lado * lado;
    }

    public double CalcularPerimetro()
    {
        return 4 * lado;
    }

    public void Dibujar()
    {
        // Código para dibujar el cuadrado
    }
}
```

Es importante recalcar, de nuevo, que esto sólo puede hacerse con interfaces, y no con clases (sería herencia múltiple, no permitida en C#). También hay que tener en cuenta que una clase puede heredar de otra y a la vez implementar una o varias interfaces. Todo esto se detalla separado por comas, tras los dos puntos al definir la clase. Por ejemplo:

```
class Cuadrado : ClaseBaseAHeredar, IFiguraGeometrica, IDibujable
{
    ...
```

6.16.3. Polimorfismo con clases abstractas o interfaces

De la misma forma que con clases normales, podemos utilizar polimorfismo con clases abstractas o con interfaces, con la salvedad de que no podremos tener objetos propios de esa clase abstracta o de la interfaz, sino siempre de sus subtipos.

Por ejemplo, basándonos en los ejemplos anteriores, podríamos tener un array de FigurasGeométricas, o de Animales, y que en cada posición hubiera un subtipo determinado.

```
IFiguraGeometrica[] figuras = new IFiguraGeometrica[10];
figuras[0] = new Cuadrado(3);
figuras[1] = new Circulo(2.5);
```

```
for(int i = 0; i < figuras.Length; i++)
{
    Console.WriteLine("Figura {0}, Area {1}, Perimetro {2}", i + 1,
    figuras[i].CalcularArea(), figuras[i].CalcularPerimetro());
}

Animal[] animales = new Animal[5];
animales[0] = new Perro("Bobby");
animales[1] = new Pato("Donald");

animales[0].Hablar();
animales[1].Hablar();
```

Observa que el tipo genérico determina lo que se puede hacer con el objeto. Si creamos un objeto *Animal* a partir de un *Pato*, no podemos llamar al método *Volar*, porque este método no es propio de la clase *Animal*, sino de su subclase *Ave*. Para poder llamar a ese método, deberíamos tener un array de aves, no de animales.

6.16.4. Clases selladas

Podemos ver las clases selladas como algo opuesto a las clases abstractas. Si una clase abstracta es aquella incompleta y de la que se debe heredar para poder construir objetos, una clase sellada es una clase de la que no se puede heredar, y que por tanto, debe ofrecer toda la funcionalidad necesaria para su propósito.

Para definir una clase sellada, se emplea el modificador *sealed* en la declaración de la clase

```
sealed class Pueba
{
    ...
}
```

En la práctica, existen algunas clases selladas que utilizamos habitualmente. Por ejemplo, la clase *String* para manejar cadenas de texto es una clase sellada. También en Java, donde el hecho de que un elemento esté sellado o sea inmutable se especifica con la palabra *final*.

6.16.4.1.Ventajas de una clase sellada

¿Qué ventajas puede tener sellar una clase? Si consultamos distintos documentos en Internet, podemos encontrar varios motivos:

- Algunos desarrolladores piensan que algunas de las clases que desarrollan son *demasiado bonitas como para ser heredadas*, dando a entender que toda la funcionalidad que puede aportar la clase ya la lleva incorporada.

- Los elementos sellados permiten mejorar la velocidad de ejecución, al no tener que estar pendiente de comprobación de subtipos y posibles polimorfismos a partir de esa clase.

6.17. Proyectos completos propuestos

La mejor forma de aplicar todos los conocimientos sobre clases y de ponerlos a prueba es crear algún proyecto de una cierta complejidad, que incluya diferentes clases, herencia, arrays de objetos, comunicación entre dichos objetos, etc. Aquí tienes unos cuantos ejercicios propuestos, sin solución (y para los que además no existirá una solución única):

(6.17.1) Crea un proyecto "Space Invaders" con todas las clases que vimos al principio del tema 6. El proyecto no será jugable, pero deberá compilar correctamente.
(6.17.2) Crea un proyecto "PersonajeMovil", que será un esqueleto sobre el que se podría ampliar para crear un juego de plataformas. Existirá una pantalla de bienvenida, una pantalla de créditos y una partida, en la que el usuario podrá mover a un personaje (que puede ser representado por una letra X). Las teclas de movimiento quedan a tu criterio, pero podrían ser "wasd" o las flechas del cursor. Si quieres que el movimiento sea más suave, puedes investigar sobre Console.Readkey (que posiblemente habrás usado si has hecho los ejercicios propuestos del proyecto ConsoleInvaders) como alternativa a Console.ReadLine.
(6.17.3) Crea un proyecto "Laberinto", a partir del proyecto "PersonajeMovil", añadiendo tres enemigos que se muevan de lado a lado de forma autónoma y un laberinto de fondo, cuyas paredes no se puedan "pisar" (la clase Laberinto puede tener métodos que informen sobre si el jugador podría moverse a ciertas coordenadas X e Y). Si el personaje toca a un enemigo, acabará la partida y se regresará a la pantalla de bienvenida.
(6.17.4) Crea un proyecto "Laberintos", a partir del proyecto "Laberinto", añadiendo premios que recoger, tres vidas para nuestro personaje y varios laberintos distintos que recorrer.
(6.17.5) Crea un proyecto "SistemaDomotico", que simule un sistema domótico para automatizar ciertas funciones en una casa: apertura y cierre de ventanas y puertas, encendido de calefacción, etc. Además de esos elementos físicos de la casa, también existirá un panel de control, desde el que el usuario podría controlar el resto de elementos, así como programar el comportamiento del sistema (por ejemplo, subir una persiana inmediatamente o hacer que la calefacción se encienda todos los días desde las 19h hasta las 23h, a una cierta temperatura).

(6.17.6) Completa el proyecto "Libro" (ejercicio 6.13.3), para que las clases de datos tengan un método "Contiene", que reciba un texto y devuelva el valor booleano "true" cuando ese texto esté contenido en el título o en el autor (y en la procedencia, para los artículos). El Main deberá permitir al usuario realizar búsquedas, aprovechando esta nueva funcionalidad.

(6.17.7) Añade al proyecto "Libro" cualquier otra funcionalidad que te parezca interesante, como la de modificar datos, borrarlos u ordenarlos.

(6.17.8) Crea en el proyecto "Libro" una clase ListaDeDocumentos, que oculte a Main los detalles de que internamente se trata de un array: deberá permitir opciones como añadir un documento, obtener los datos de uno de ellos, buscar entre la lista de todos ellos, etc. Los criterios de diseño quedan a tu criterio. Por ejemplo, puedes hacer que la búsqueda devuelve un array con los números de ficha encontrados, o un array con el contenido de esas fichas, o el contenido de la siguiente ficha que cumpla con esos criterios (en ese caso, quizá te interese distinguir entre un método "BuscarPrimero", que localice la primera aparición, y otro "BuscarSiguiente" que trate de encontrar a partir de la posición actual).

(6.17.9) Amplía el esqueleto de ConsoleInvaders (6.14.4), con una nueva clase "BloqueDeEnemigos", que será la que contenga el array de enemigos, de modo que se simplifique la lógica de la clase Partida. Esta clase tendrá un método Dibujar, que mostrará todo el array en pantalla, y un método Mover, que moverá todo el bloque hacia la derecha y la izquierda de forma alternativa (deberás comprobar la posición inicial del primer enemigo y la posición final del último enemigo). En cada pasada por el bucle de juego deberás llamar a Mover, de modo que los enemigos permanecerán quietos si nuestra nave no se mueve, y se moverán cuando pulsemos las teclas para desplazar nuestra nave.

(6.17.10) Amplía la versión 6.17.9 de ConsoleInvaders para que los enemigos se muevan "solos". La forma es no hacer siempre "Console.ReadKey();", que deja el programa bloqueado hasta que se pulse una tecla, sino mirar el teclado sólo cuando haya teclas disponibles, lo que se puede saber con "if (Console.KeyAvailable)" (tienes más detalles en el apartado 12.2). El juego irá demasiado rápido si no fuerzas una pausa (por ejemplo, de 100 milisegundos) al final de cada pasada por el bucle principal, para lo que puedes usar "Thread.Sleep(100);" (que necesita al principio del programa "using System.Threading;" y que veremos con más detalle más adelante.

(6.17.11) Crea una clase Disparo en ConsoleInvaders. Cuando el usuario pulse cierta tecla (Espacio, por ejemplo), aparecerá un disparo encima de la nave, y se moverá hacia arriba hasta que desaparezca por la parte superior de la pantalla. Existirá un único disparo, y no se podrá volver a disparar si está activo (en pantalla). Inicialmente estará desactivado, y lo volverá a estar cuando llegue al margen de la pantalla.

(6.17.12) En ConsoleInvaders, crea un método "ColisionaCon" en la clase Sprite, que reciba otro Sprite como parámetro y devuelva el valor booleano "true" si ambos sprites están "chocando" o "false" en caso contrario. Tendrás que pensar qué relación habrá entre las coordenadas X e Y de ambos sprites para que "choquen".
(6.17.13) En ConsoleInvaders, crea una clase Ovni, un nuevo tipo de Enemigo que no estará siempre activo. Su método Mover hará que se mueva hacia la derecha si ya está activo o, en caso contrario, que genere un número al azar para decidir si debe activarse.
(6.17.14) En ConsoleInvaders, comprueba colisiones entre el disparo y el Ovni. Si hay colisión, desaparecerán ambos y el jugador obtendrá 50 puntos.
(6.17.15) En ConsoleInvaders, comprueba colisiones entre el disparo y el bloque de enemigos. Si el disparo toca algún enemigo, ambos desaparecerán y el jugador obtendrá 10 puntos.
(6.17.16) En ConsoleInvaders, añade una clase "Marcador", que muestre la puntuación y la cantidad de vidas restantes (que por ahora, siempre será 3).
(6.17.17) En ConsoleInvaders, añade la posibilidad de que algunos enemigos al azar puedan disparar (sus disparos "van hacia abajo"; piensa si crear una nueva clase o ampliar las funcionalidades de la clase Disparo que ya tienes). Ajusta la frecuencia de disparos, de modo que el juego continúe siendo jugable. Si uno de esos disparos impacta con la nave, se perderá una vida y el disparo desaparecerá. Si se pierden las 3 vidas, acaba la partida y se volverá a la pantalla de presentación.
(6.17.18) En ConsoleInvaders, crea la estructura que sea necesaria para almacenar las "mejores puntuaciones" (por ejemplo un array de enteros, o de "struct" si también vas a pedir su nombre al usuario), que se actualizarán al terminar cada partida y se mostrarán en la pantalla de bienvenida.
(6.17.19) En ConsoleInvaders, añade las "torres defensivas", que protegen al jugador y que se van rompiendo poco a poco cada vez que un disparo (del jugador o de los enemigos) impacta con ellas.