6. Programación orientada a objetos

6.1. ¿Por qué los objetos?

Hasta ahora hemos estado "cuadriculando" todo para obtener algoritmos: tratábamos de convertir cualquier cosa en una función que pudiéramos emplear en nuestros programas. Cuando teníamos claros los pasos que había que dar, buscábamos las variables necesarias para dar esos pasos.

Pero no todo lo que nos rodea es tan fácil de cuadricular. Supongamos por ejemplo que tenemos que introducir datos sobre una puerta en nuestro programa. ¿Nos limitamos a programar los procedimientos AbrirPuerta y CerrarPuerta? Al menos, deberíamos ir a la zona de declaración de variables, y allí guardaríamos otras datos como su tamaño, color, etc.

No está mal, pero es "antinatural": una puerta es un conjunto: no podemos separar su color de su tamaño, o de la forma en que debemos abrirla o cerrarla. Sus características son tanto las físicas (lo que hasta ahora llamábamos variables) como sus comportamientos en distintas circunstancias (lo que para nosotros eran las funciones). Todo ello va unido, formando un "**objeto**".

Por otra parte, si tenemos que explicar a alguien lo que es el portón de un garaje, y ese alguien no lo ha visto nunca, pero conoce cómo es la puerta de su casa, le podemos decir "se parece a una puerta de una casa, pero es más grande para que quepan los coches, está hecha de metal en vez de madera...". Es decir, podemos describir unos objetos a partir de lo que conocemos de otros.

Finalmente, conviene recordar que "abrir" no se refiere sólo a una puerta. También podemos hablar de abrir una ventana o un libro, por ejemplo.

Con esto, hemos comentado casi sin saberlo las tres características más importantes de la Programación Orientada a Objetos (OOP):

- **Encapsulación**: No podemos separar los comportamientos de las características de un objeto. Los comportamientos serán funciones, que en OOP llamaremos **métodos**. Las características de un objeto serán variables, como las que hemos usado siempre (las llamaremos **atributos**). La apariencia de un objeto en C#, como veremos un poco más adelante, recordará a un registro o "struct".

- **Herencia**: Unos objetos pueden heredar métodos y datos de otros. Esto hace más fácil definir objetos nuevos a partir de otros que ya teníamos anteriormente (como ocurría con el portón y la puerta) y facilitará la reescritura de los programas, pudiendo aprovechar buena parte de los anteriores... si están bien diseñados.

- **Polimorfismo**: Un mismo nombre de un método puede hacer referencia a comportamientos distintos (como abrir una puerta o un libro). Igual ocurre para los datos: el peso de una puerta y el de un portón los podemos llamar de igual forma, pero obviamente no valdrán lo mismo.

Otro concepto importante es el de "clase": **Una clase** es un conjunto de objetos que tienen características comunes. Por ejemplo, tanto mi puerta como la de mi vecino son puertas, es decir, ambas son objetos que pertenecen a la clase "puerta". De igual modo, tanto un Ford Focus como un Honda Civic o un Toyota Corolla son objetos concretos que pertenecen a la clase "coche".

6.2. Objetos y clases en C#

Vamos con los detalles. Las **clases en C#** se definen de forma parecida a los registros (struct), sólo que ahora también incluirán funciones. Así, la clase "Puerta" que mencionábamos antes se podría declarar así:

```
public class Puerta
{

  int ancho;     // Ancho en centimetros
  int alto;      // Alto en centimetros
  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
```

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 (y además, nos pueden "mostrar su estado, para comprobar que todo funciona correctamente).

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"):

```
/*---------------------------*/
/*  Ejemplo en C# nº 59:     */
/*  ejemplo59.cs             */
/*                           */
/*  Primer ejemplo de clases */
/*                           */
/*  Introduccion a C#,       */
/*    Nacho Cabanes          */
/*---------------------------*/

using System;

public class Puerta
{

  int ancho;     // Ancho en centimetros
  int alto;      // Alto en centimetros
  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


public class Ejemplo59
{

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

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

    Console.WriteLine("\nVamos 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 son los nuevos objetos con los que vamos a practicar.
- La clase "Ejemplo59", que representa a nuestra aplicación.

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#, al contrario que en otros 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, o "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", 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, desde Ejemplo59).
- 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 su uso.

Normalmente, como forma de ocultar datos, crearemos funciones auxiliares GetXXX y SetXXX que permitan acceder al valor de los atributos (en C# existe una forma alternativa de hacerlo, usando "propiedades", que veremos más adelante):

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

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

```
}
```

También puede desconcertar que en "Main" aparezca la palabra "**static**", mientras que no lo hace en los métodos de la clase "Puerta". Veremos este detalle un poco más adelante, en el tema 7, pero de momento perderemos la costumbre de escribir "static" antes de cada función: a partir de ahora, sólo Main lo será.

Ejercicio propuesto:

- **(6.2.1)** Crear 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. Crear también una clase llamada PruebaPersona. Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona, les asignará un nombre y les pedirá que saluden.

En un proyecto grande, es recomendable que cada clase esté en su propio fichero fuente, de forma que se puedan localizar con rapidez (en los que hemos hecho en el curso hasta ahora, no era necesario, porque eran muy simples). Precisamente por eso, es interesante (pero no obligatorio) que cada clase esté en un fichero que tenga el mismo nombre: que la clase Puerta se encuentre en el fichero "Puerta.cs". Esta es una regla que no seguiremos en algunos de los ejemplos del texto, por respetar la numeración consecutiva de los ejemplos, pero que sí se debería seguir en un proyecto de mayor tamaño, formado por varias clases.

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

```
gmcs fuente1.cs fuente2.cs fuente3.cs
```

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

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

(Un poco más adelante veremos cómo crear un programa formado por varios fuentes usando Visual Studio y otros entornos integrados).

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

```
/*--------------------------*/
/*  Ejemplo en C# nº 59b:   */
/*  ejemplo59b.cs           */
/*                          */
/*  Dos clases en dos       */
/*  ficheros (fichero 1)    */
```

```
/*                          */
/*  Introduccion a C#,      */
/*    Nacho Cabanes         */
/*--------------------------*/

using System;

public class Puerta
{

  int ancho;     // Ancho en centimetros
  int alto;      // Alto en centimetros
  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
```

Y la segunda clase podría ser:

```
/*--------------------------*/
/*  Ejemplo en C# nº 59c:   */
/*  ejemplo59c.cs           */
/*                          */
/*  Dos clases en dos       */
/*  ficheros (fichero 2)    */
/*                          */
/*  Introduccion a C#,      */
/*    Nacho Cabanes         */
/*--------------------------*/

public class Ejemplo59c
{

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

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

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

}
```

Y lo compilaríamos con:

```
gmcs ejemplo59b.cs ejemplo59c.cs -out:ejemplo59byc.exe
```

Aun así, para estos proyectos formados por varias clases, lo ideal es usar algún entorno más avanzado, como SharpDevelop o VisualStudio, que permitan crear todas las clases con comodidad, saltar de una clase a clase otra rápidamente, que marquen dentro del propio editor la línea en la que están los errores... por eso, al final de este tema tendrás un apartado con una introducción al uso de SharpDevelop.

Ejercicio propuesto:

- **(6.2.2)** Modificar el fuente del ejercicio anterior, para dividirlo en dos ficheros: Crear 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. Crear también una clase llamada PruebaPersona, en el fichero "pruebaPersona.cs". Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona, les asignará un nombre y les pedirá que saluden.

6.3. La herencia. Visibilidad

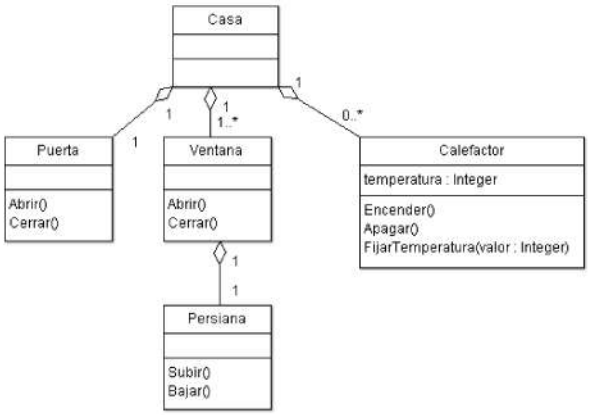
Vamos a ver ahora cómo definir una nueva clase de objetos a partir de otra ya existente. Por ejemplo, vamos a crear una clase "Porton" a partir de la clase "Puerta". Un portón tendrá las mismas características que una puerta (ancho, alto, color, abierto o no), pero además se podrá bloquear, lo que supondrá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

```
public class Porton: Puerta
{

  bool bloqueada;

  public void Bloquear()
  {
    bloqueada = true;
  }

  public void Desbloquear()
  {
    bloqueada = false;
```



Este diagrama es una forma más simple de ver las clases existentes y las relaciones entre ellas. Si generamos las clases a partir del diagrama, tendremos parte del trabajo hecho: ya "sólo" nos quedará rellenar los detalles de métodos como "Abrir", pero el esqueleto de todas las clases ya estará "escrito" para nosotros.

Ejercicios propuestos:

- **(6.4.1)** Crear las clases correspondientes al diagrama de clases del apartado 6.4, con sus correspondientes métodos y atributos.

6.5. Constructores y destructores.

Hemos visto que al declarar una clase, 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 declarando un "**constructor**" para la clase.

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

Se declara usando el mismo nombre que el de la clase, 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 usara estos dos constructores para crear dos puertas con características iniciales distintas podría ser:

```
/*----------------------------*/
/*  Ejemplo en C# nº 61:      */
/*  ejemplo61.cs              */
/*                            */
/*  Tercer ejemplo de clases  */
/*  Constructores             */
/*                            */
/*  Introduccion a C#,        */
/*    Nacho Cabanes           */
/*----------------------------*/

using System;

public class Puerta
{

  int ancho;     // Ancho en centimetros
  int alto;      // Alto en centimetros
  int color;     // Color en formato RGB
  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


public class Ejemplo61
{

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

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

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

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

}
```

Nota: al igual que existen los "constructores", también podemos indicar 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 sabemos).

Un "**destructor**" se llama igual que la clase, pero precedido por el símbolo "~", no tiene tipo de retorno, y no necesita ser "public", como ocurre en este ejemplo:

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

7. Utilización avanzada de clases

7.1. La palabra "static"

Desde un principio, nos hemos encontrado con que "Main" siempre iba acompañado de la palabra "static". 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 valer 4 para cualquier objeto que pertenezca a esa clase (cualquier coches). Por eso, se podría declarar como "static".

De igual modo, 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, si queremos que se pueda usar la función "BorrarPantalla" de una clase "Hardware" sin necesidad de crear primero un objeto perteneciente a esa clase, lo podríamos conseguir así:

```
public class Hardware
{
    ...

    public static void BorrarPantalla ()
    {
      ...
    }
```

que desde dentro de "Main" (incluso perteneciente a otra clase) se usaría con el nombre de la clase delante:

```
public class Juego
{
    ...

    public  ComienzoPartida()    {
      Hardware.BorrarPantalla ();
      ...
```

Desde una función "static" no se puede llamar a otras funciones que no lo sean. Por eso, como nuestro "Main" debe ser static, deberemos siempre elegir entre:

- Que todas las demás funciones de nuestro fuente también estén declaradas como "static", por lo que podrán ser utilizadas desde "Main" (como hicimos en el tema 5).
- Declarar un objeto de la clase correspondiente, y entonces sí podremos acceder a sus métodos desde "Main":

- **(7.4.2)** Crea una versión ampliada del ejercicio 7.4.1, en la que el constructor de todas las clases hijas se apoye en el de la clase "Trabajador".

7.5. La palabra "this": el objeto actual

Podemos hacer referencia al objeto que estamos usando, con "this":

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

En muchos casos, podemos evitarlo. Por ejemplo, normalmente es preferible usar otro nombre en los parámetros:

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

Y en ese uso 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, int y )
    : this (x)
{
    fila = y;
    // Pasos adicionales
}
```

Y se usa mucho para que unos objetos "conozcan" a los otros:

```
public void IndicarEnemigo (ElemGrafico enemigo)
{
   ...
```

De modo que el personaje de un juego le podría indicar al adversario que él es su enemigo con

```
miAdversario.IndicarEnemigo(this);
```