```
    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;
```

```
  }
}
```

Con "public 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.

No tenemos por qué heredar todo; 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:

```
  public new void MostrarEstado()
  {
    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);
  }
```

Aun así, esto todavía no funciona: los atributos de una Puerta, como el "ancho" y el "alto" estaban declarados como "privados" (es lo que se considera si no decimos los contrario), por lo que no son accesibles desde ninguna otra clase, ni siquiera desde Porton.

La solución más razonable no es declararlos como "public", porque no queremos que sean accesibles desde cualquier sitio. Sólo querríamos que esos datos estuvieran disponibles para todos los tipos de Puerta, incluyendo sus "descendientes", como un Porton. Esto se puede conseguir usando otro método de acceso: "**protected**". Todo lo que declaremos como "protected" será accesible por las clases derivadas de la actual, pero por nadie más:

```
public class Puerta
{

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

  public void Abrir()
  ...
```

(Si quisiéramos dejar claro que algún elemento de una clase debe ser totalmente privado, podemos usar la palabra "**private**", en vez de "public" o "protected").

Un 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 en C# nº 60:     */
/*  ejemplo60.cs             */
/*                           */
/*  Segundo ejemplo de       */
/*  clases: herencia         */
/*                           */
/*  Introduccion a C#,       */
/*    Nacho Cabanes          */
/*--------------------------*/

using System;

// -------------------------------
public class Puerta
{

  protected int ancho;     // Ancho en centimetros
  protected int alto;      // Alto en centimetros
  protected int color;     // Color en formato RGB
  protected 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 Porton: Puerta
{

  bool bloqueada;

  public void Bloquear()
  {
    bloqueada = true;
  }

  public void Desbloquear()
  {
    bloqueada = false;
  }

  public new void MostrarEstado()
```

```
  {
    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);
  }

} // Final de la clase Porton

// ------------------------------
public class Ejemplo60
{

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

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

    Console.WriteLine("\nVamos a bloquear...");
    p.Bloquear();
    p.MostrarEstado();

    Console.WriteLine("\nVamos a desbloquear y a abrir...");
    p.Desbloquear();
    p.Abrir();
    p.MostrarEstado();
  }

}
```

Ejercicios propuestos:

- **(6.3.1)** Ampliar las clases del ejercicio 6.2.2, creando un nuevo proyecto con las siguientes características: 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". Crear también una clase llamada Prueba-Persona2, 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.
- **(6.3.2)** Ampliar las clases del ejercicio 6.3.1, 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". Esta clase deberá heredar las características de la clase "Persona", pero redefinir el método "Saludar", para que escriba en pantalla "Ciao". Crear también una clase llamada PruebaPersona3, en el fichero " pruebaPersona3.cs". Esta clase deberá contener sólo la función Main, que creará un objeto de tipo Persona, dos de tipo PersonaInglesa, uno de tipo PersonaItaliana, les asignará un nombre, les pedirá que saluden y pedirá a la persona inglesa que tome té.

Ejercicios propuestos:

- **(6.5.1)** Ampliar las clases del ejercicio 6.4.1, para que todas ellas contengan constructores. Los constructores de casi todas las clases estarán vacíos, excepto del de "Calefactor", que prefijará una temperatura de 25 grados.

6.6. Polimorfismo y sobrecarga

Esos dos constructores "Puerta()" y "Puerta(int ancho, int alto)", que se llaman igual pero reciben distintos parámetros, y se comportan de forma que puede ser distinta, son ejemplos de "**polimorfismo**" (funciones que tienen el mismo nombre, pero distintos parámetros, y que quizá no se comporten de igual forma).

Un concepto muy relacionado con el polimorfismo es el de "**sobrecarga**": dos funciones están sobrecargadas cuando se llaman igual, reciben el mismo número de parámetros, pero se aplican a objetos distintos, así:

```
puerta.Abrir ();
libro.Abrir ();
```

En este caso, la función "Abrir" está sobrecargada: se usa tanto para referirnos a abrir un libro como para abrir una puerta. Se trata de dos acciones que no son exactamente iguales, que se aplican a objetos distintos, pero que se llaman igual.

Ejercicios propuestos:

- **(6.6.1)** Añade a la clase "Ventana" un nuevo método Abrir, que reciba un parámetro, que será un número del 0 al 100 que indique hasta qué punto se debe abrir la ventana (100=100%, abierta; 0=0%, cerrada). Crea los atributos auxiliares que necesites para reflejar esa información.

6.7. 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 en C# nº 62:    */
/*  ejemplo62.cs            */
/*                          */
/*  Cuarto ejemplo de clases */
/*  Constructores y herencia */
/*                          */
/*  Introduccion a C#,      */
/*    Nacho Cabanes         */
/*--------------------------*/
```

```
using System;

public class Animal
{

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

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

public class Perro: Animal
{

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

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

public class Gato: Animal
{

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


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

public class GatoSiames: Gato
{

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

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

public class Ejemplo62
{

  public static void Main()
  {
    Animal a1     = new Animal();
    GatoSiames a2 = new GatoSiames();
    Perro a3      = new Perro();
    Gato a4       = new Gato();
  }

}
```

- **(7.3.2)** Crea una variante del ejercicio 7.3.1, en la que se cree un único array "de trabajadores", que contenga un objeto de cada clase.
- **(7.3.3)** Amplía el ejercicio 7.3.2, para añadir un método "Hablar" en cada clase, redefiniéndolo en las clases hijas usando "new" y probándolo desde "Main".
- **(7.3.4)** Crea una variante del ejercicio 7.3.3, que use "override" en vez de "new".

7.4. 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, 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 en C# nº 67:    */
/*  ejemplo67.cs            */
/*                          */
/*  Ejemplo de clases       */
/*  Llamar a la superclase  */
/*                          */
/*  Introduccion a C#,      */
/*    Nacho Cabanes         */
/*--------------------------*/

using System;

public class Animal
{

}

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

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


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

```
public class GatoSiames: Gato
{

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

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

public class Ejemplo67
{

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

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

  }

}
```

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, no en el que tiene x e y como parámetros).

Ejercicios propuestos:

- **(7.4.1)** Crea una versión ampliada del ejercicio 7.3.4, en la que el método "Hablar" de todas las clases hijas se apoye en el de la clase "Trabajador".