Tema 8. Herencia y Polimorfismo

1. Introducción

La herencia es uno de los conceptos más importantes de la Programación Orientada a Objetos (POO) y uno de los fundamentos sobre los cuales está construido C#.

Herencia se refiere a la capacidad de crear clases que **heredan** ciertos aspectos o funcionalidades de otras clases primarias.

Todo el framework de .NET se basa en el concepto de Herencia y por esto en .NET "todo es un objeto". Incluso un simple número es una instancia de una clase que se hereda de la clase System.Object.

2. Utilizando la herencia.

En C# declaramos que una clase hereda de otra con la siguiente sintaxis:

```
public class ClaseDerivada: ClaseBase
{
      …
}
```

La clase hereda de la **clase base,** y los miembros, propiedades y métodos de la clase base pasan a formar parte ("se heredan"), de la **clase derivada**.

Existen varias formas de llamar a las clases de las que se hereda y las que heredan: superclase y subclase, clase padre y clase hija, clase base y clase derivada.

Vamos a ir **desarrollando un ejemplo** en el que trabajemos con clases bases y clases derivadas.

Imaginemos que queremos hacer un programa que trabaje con **formas geométricas**.

Vamos a hacer una clase Figura (repasad en el tema anterior cómo crear una nueva clase) que tendrá los datos comunes a distintas figuras y luego haremos clases para cada tipo de figura.

Podríamos tener una clase **Figura** así:

```
public class Figura
{
    private int mPosicionX, mPosicionY;
    private string mColor;

    public string Color
    {
        get { return mColor; }
        set { mColor = value; }
    }
```

```
        // Constructor de la figura
        public Figura (int x, int y, string color)
        {
            mPosicionX = x;
            mPosicionY = y;
            mColor = color;
        }

    }
```

Y podríamos tener una clase Circulo que de momento heredaría de la clase Figura:

```
    public class Circulo : Figura
    {
        private int mRadio;

        public int Radio
        {
            get { return mRadio; }
            set { mRadio = value; }
        }

        public Circulo(int x, int y, string color, int radio) : base(x, y, color)
        {
            mRadio = radio;
        }
    }
```

Accesibilidad.

Fijémonos en varios **detalles**:

Lo que estamos haciendo es que la clase Circulo **hereda la funcionalidad** de la clase Figura.

La clase Circulo **puede añadir** los miembros, propiedades o métodos que le interesen, para aumentar la funcionalidad de la clase Figura. Por ejemplo, añade el miembro mRadio.

El constructor de la clase Circulo llama al constructor de la clase base, para que además de inicializar sus miembros (mRadio), inicialice los miembros de la clase base (mPosicionX, mPosicionY, mColor).

```
        public Circulo(int x, int y, string color, int radio) : base(x, y, color)
```

Por otro lado, todo lo que es **público** en la clase base, también es público en la clase derivada.

Así nosotros podríamos, al declarar un objeto de tipo Circulo, acceder a su propiedad Radio, pero también a su propiedad heredada, Color. Por ejemplo, en un botón de formulario podríamos hacer:

```
        private void button1_Click(object sender, EventArgs e)
        {
            Circulo circulo = new Circulo(10, 20, "Azul", 5);

            circulo.Color = "Rojo";
            circulo.Radio = 10;

        }
```

No podríamos, sin embargo, acceder a mPosicionX, ni mPosicionY porque son miembros **privados**. De hecho, no podemos acceder a ellos ni siquiera dentro de la clase heredada Circulo.

Si quisiéramos poder acceder a ellos desde las clases heredadas tendríamos que haber utilizado el modificador protected:

```
protected int mPosicionX, mPosicionY;
```

De esta forma sí que podríamos acceder a ellos en las clases que hereden de Figura.

En resumen, si un miembro, método o propiedad:

- Es declarado **private**, únicamente se puede acceder ellos dentro de la propia clase.
- Es declarado **public**, se puede acceder ellos desde fuera de la clase, haciendo objeto.método (por ejemplo).
- Es declarado **protected**, no se puede acceder al mismo desde fuera de la clase, pero sí pueden acceder a ellos las clases que hereden de la clase base.

Declaración de nuevos métodos

Hemos visto que una clase hija (o derivada) puede utilizar las propiedades y miembros de la clase padre. También puede utilizar los métodos.

Imaginemos que hacemos un método en Figura que devuelve qué clase es:

```
    public class Figura
    {
       ...
       ...

        public string QuienSoy()
        {
            return "Soy una figura";
        }

    }
```

Si nosotros llamamos a ese método con un objeto de tipo Circulo, nos dirá que "Soy una figura."

```
        private void button1_Click(object sender, EventArgs e)
        {
            Circulo circulo = new Circulo(10, 20, "Azul", 5);

            MessageBox.Show(circulo.QuienSoy());

        }
```

Pero las clases heredadas pueden **redefinir** los métodos:

```
    public class Circulo : Figura
    {
       ...
       ...

        new public string QuienSoy()
        {
            return "Soy un círculo.";
        }
    }
```

La palabra reservada **new** indica al compilador que estamos **redefiniendo** un método heredado. Si no la ponemos el programa compila, pero tendremos un warning (aviso).

Si ahora ejecutamos lo del formulario, nos dirá "Soy un círculo", en vez de "Soy una figura".

3. Declaración de métodos virtuales y sobreescritura de métodos.

Con lo que hemos hecho en el apartado anterior, lo que conseguimos es hacer una "**nueva**" función QuienSoy.

Pero esta **no es la manera correcta** de encarar esa situación en POO.

Para ello es mejor utilizar **métodos virtuales** en las clases base y sobreescribirlos (override) en las clases heredadas.

Esto es así para permitir el **polimorfismo**.

El polimorfismo consiste en que podemos hacer que una instancia de una clase base "apunte" a un objeto de una clase derivada.

Esto nos permite hacer cosas como:

```
Circulo circulo = new Circulo(10, 20, "Azul", 5);

Figura figura = circulo;
```

¿Por qué es interesante el **polimorfismo**? Estamos adelantando conceptos, pero la idea es que, si por ejemplo tenemos la clase Circulo y la clase Cuadrado, y quisiera tener listas de elementos, no me haría falta tener una lista para los círculos y otra para los cuadrados, sino que podría tener una lista de figuras (y en ella tener círculos y cuadrados).

Tranquilos. Lo veremos más adelante.

El problema que se nos plantea ahora es que si yo hago algo como esto:

```
Circulo circulo = new Circulo(10, 20, "Azul", 5);

MessageBox.Show(circulo.QuienSoy());

Figura figura = circulo;
MessageBox.Show(figura.QuienSoy());
```

el segundo mensaje me dice que soy una figura, cuando realmente soy un círculo y me puede interesar decir que soy un círculo. Esto es así porque QuienSoy de Circulo es una **nueva** función (no sobreescribe).

Esto se soluciona mediante la utilización de **funciones virtuales (virtual)** en las clases bases y sobreescribir (**override**) estas funciones en las clases derivadas:

```
public class Figura
{
   ...
   ...

    public virtual string QuienSoy()
    {
        return "Soy una figura";
    }

}
```

y sobrescribimos en Circulo la función:

```
public class Circulo : Figura
{
   ...
   ...

    public override string QuienSoy()
    {
        return "Soy un círculo.";
    }
}
```

Si ahora hacemos lo de antes:

```
            Circulo circulo = new Circulo(10, 20, "Azul", 5);

            MessageBox.Show(circulo.QuienSoy());

            Figura figura = circulo;
            MessageBox.Show(figura.QuienSoy());
```

Ambos mensajes me dicen ahora que soy un círculo. Esto es así porque QuienSoy de Circulo es una función que hemos **sobrescrito**.

Incluso podemos llamar a la función QuienSoy de la clase base en la clase derivada:

```
public class Circulo : Figura
{
   ...
   ...

    public override string QuienSoy()
    {
        string texto = base.QuienSoy();
        texto = texto + "\nSoy un círculo.";

        return texto;
    }
}
```

En este caso lo del formulario nos diría Soy una figura. Soy un círculo.

Sobreescritura del método ToString()

En temas anteriores hemos utilizado el método ToString() para conseguir el texto de un entero, por ejemplo.

Realmente ToString es un método virtual de la clase System.Object, de la cual heredan todas las clases de C#. Podemos, por tanto, **sobrescribirla** en las clases que nos interese.

Por ejemplo, en la clase Figura:

```
public class Figura
{
   ...
   ...

   public override string ToString()
    {
        string texto;

        texto = "Posición X: " + mPosicionX +
                "\nPosición Y: " + mPosicionY +
                "\nColor: " + mColor;

        return texto;
    }
}
```

Y volverla a sobreescribir en Círculo (utilizando ToString de Figura):

```
public class Circulo : Figura
{
   ...
   ...

    public override string ToString()
    {
        string texto = base.ToString();
        texto = texto + "\nRadio: " + mRadio;

        return texto;
    }
}
```

Podemos hacer ahora por ejemplo MessageBox.Show(circulo.ToString());

4. Clases y métodos abstractos

Vamos a estudiar a continuación el concepto de **clase abstracta**. Una clase abstracta es una clase de la cual no van a existir objetos, pero sí que va a implementar cosas que luego se van a utilizar en las clases que deriven de ella.

Por ejemplo, en el programa que vamos a desarrollar nosotros no vamos a crear objetos de tipo Figura. Nuestros objetos serán círculos, cuadrados …

Por tanto, nuestra clase será **abstracta**. Habrá clases que hereden de ella, pero **no tendremos objetos** de la misma.

La forma de hacer abstracta una clase es con la palabra reservada **abstract**:

```
public abstract class Figura
{
      ...
}
```

Las clases abstractas nos permiten declarar **métodos abstractos**. Los métodos abstractos son aquellos que **no tienen cuerpo**, porque luego se van a reescribir completamente en las clases hijas.

Por ejemplo, podemos tener en Figura un método abstracto Area().

Una figura no tiene forma de calcular el área, pero esa función abstracta se **redefinirá** en los círculos, cuadrados … que sí tienen área.

```
public abstract class Figura
{
   ...
   ...

    public abstract double Area();
}
```

En nuestra clase Circulo reescribimos ese método Area. De hecho, es **imprescindible** que lo reescribamos. En otro caso, el compilador nos da un error.

```
public class Circulo : Figura
{
   ...
   ...

    public override double Area()
    {
        return Math.PI * mRadio * mRadio;
    }
}
```

Así podemos luego llamar al método Area en el formulario:

```
private void button1_Click(object sender, EventArgs e)
{
    Circulo circulo = new Circulo(10, 20, "Azul", 5);

    MessageBox.Show("El area es: " + circulo.Area());
}
```

Es importante que entendamos que **no podemos** crear objetos de tipo Figura:

```
    Figura figura = new Figura(10, 20, "Azul");
```

Esta línea nos da un error de compilación, porque Figura es una clase abstracta.

Sin embargo, sí podemos hacer que una figura "apunte" a un círculo:

```
Figura figura = circulo;
```

algo que es **importante para el polimorfismo** como veremos más adelante.

Si hacemos:

```
    Circulo circulo = new Circulo(10, 20, "Azul", 5);

    Figura figura = circulo;

    MessageBox.Show("El area es: " + figura.Area());
```

El mensaje me mostrará el área del círculo al cual apunta la figura.

5. Recapitulamos.

Tenemos dos clases. Una clase base abstracta llamada Figura y otra clase llamada Circulo que hereda de la clase Figura.

La clase abstracta Figura quedaría así:

```
public abstract class Figura
{
    // Miembros de la clase
    private int mPosicionX, mPosicionY;
    string mColor;

    // Propiedades de la clase
    public int PosicionX
    {
        get { return mPosicionX; }
        set { mPosicionX = value; }
    }

    public int PosicionY
    {
        get { return mPosicionY; }
        set { mPosicionY = value; }
    }

    public string Color
    {
        get { return mColor; }
        set { mColor = value; }
    }

    // Constructor de la figura
    public Figura (int x, int y, string color)
    {
        mPosicionX = x;
        mPosicionY = y;
        mColor = color;
    }

    // Método virtual QuienSoy. Se puede reescribir en las clases que heredan
    public virtual string QuienSoy()
    {
        return "Soy una figura";
    }

    // Sobreescritura del método ToString que heredamos de System.Object
    public override string ToString()
    {
        string texto;

        texto = "Posición X: " + mPosicionX +
                "\nPosición Y: " + mPosicionY +
                "\nColor: " + mColor;
```

```
            return texto;
        }

        // Método abstracto Area. No tiene cuerpo.
        // Se reescribe en las clases que heredan de Figura
        public abstract double Area();

    }
```

La clase Circulo que hereda de Figura:

```
    public class Circulo : Figura
    {
        // Miembros privados
        private int mRadio;

        public int Radio
        {
            get { return mRadio; }
            set { mRadio = value; }
        }

        // Constructor. Llama al constructor de Figura
        public Circulo(int x, int y, string color, int radio) : base(x, y, color)
        {
            mRadio = radio;
        }



        // Sobreescritura del método virtual QuienSoy
        public override string QuienSoy()
        {
            return "Soy un círculo.\n";
        }


        // Sobreescritura de ToString
        public override string ToString()
        {
            string texto = base.ToString();
            texto = texto + "\nRadio: " + mRadio;

            return texto;
        }

        // Sobreescritura del método abstracto Area
        public override double Area()
        {
            return Math.PI * mRadio * mRadio;
        }
    }
```

6. Creación de más clases que heredan. La clase Cuadrado

Con lo que hemos visto hasta ahora realmente podríamos haber hecho todo dentro de la misma clase Circulo. Es decir, hasta el momento la herencia no tiene más sentido que haber aprendido como crear clases heredadas.

Lo que vamos a hacer a continuación es crear una **nueva clase Cuadrado** que también herede de Figura y aprovechemos funcionalidad ya implementada en esa clase.

```
class Cuadrado : Figura
{
    // Miembros privados
    private int mLado;

    public int Lado
    {
        get { return mLado; }
        set { mLado = value; }
    }

    // Constructor. Llama al constructor de Figura
    public Cuadrado(int x, int y, string color, int lado) : base(x, y, color)
    {
        mLado = lado;
    }

    // Sobreescritura del método virtual QuienSoy
    public override string QuienSoy()
    {
        return "Soy un cuadrado.\n";
    }

    // Sobreescritura de ToString
    public override string ToString()
    {
        string texto = base.ToString();
        texto = texto + "\nLado: " + mLado;

        return texto;
    }

    // Sobreescritura del método abstracto Area
    public override double Area()
    {
        return mLado * mLado;
    }
}
```

En el formulario podemos crear un objeto de tipo Cuadrado y llamar a los métodos que nos interese:

```
        private void button1_Click(object sender, EventArgs e)
        {
            Cuadrado cuadrado = new Cuadrado(20, 40, "Rojo", 10);

            MessageBox.Show(cuadrado.QuienSoy());
            MessageBox.Show(cuadrado.ToString());
            MessageBox.Show("El área es " + cuadrado.Area());
        }
```

Os recomiendo hacer en este punto el ejercicio 1 de la lista de ejercicios.

7. Polimorfismo

El **polimorfismo** es una palabra que nos dice que algo puede tener muchas formas. Ya lo hemos visto anteriormente, pero, resumiendo, aplicado a la POO y la herencia el polimorfismo hace referencia a que un objeto puede tener distintas formas.

Esto, que ya hemos visto brevemente en apartados anteriores, se consigue mediante la herencia, ya que un objeto de una clase **base**, puede "ser" en un momento dado un objeto de una clase **heredada**.

En los ejemplos que hemos estado haciendo, un objeto de clase Figura, en un momento dado puede ser un círculo y en otro puede ser un cuadrado…

```
            Circulo circulo = new Circulo(10, 20, "Azul", 5);
            Cuadrado cuadrado = new Cuadrado(20, 40, "Rojo", 10);

            // En este momento pasa a referenciar un objeto de tipo círculo
            figura = circulo;
            // En este momento pasa a referenciar un objeto de tipo cuadrado
            figura = cuadrado;
```

Como vemos un objeto de tipo Figura puede "apuntar" o referenciar a objetos de tipos distintos (Circulo y Cuadrado), porque heredan de la clase Figura.

Incluso puedo crear un objeto de Circulo en una instancia o variable de Figura:

```
        Figura figura = new Circulo(10, 20, "Verde", 3);
```

Una de las cosas importantes del polimorfismo es que cuando se ejecuten métodos virtuales o abstractos que han sido sobrescritos en una clase heredada, se ejecutarán esos métodos sobrescritos.

Es decir, si mi figura apunta a un objeto de tipo Circulo y ejecutamos QuienSoy dirá que es un círculo…ya que ejecuta el QuienSoy de la clase Circulo.

Veamos el siguiente código y el resultado de ejecutarlo:

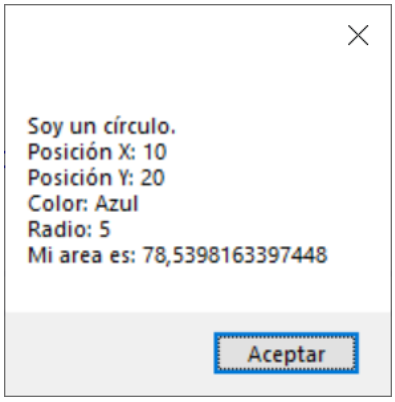
```
        private void button1_Click(object sender, EventArgs e)
        {
            Circulo circulo = new Circulo(10, 20, "Azul", 5);
            Cuadrado cuadrado = new Cuadrado(20, 40, "Rojo", 10);
            Figura figura;
            String texto;

            // En este momento pasa a referenciar un objeto de tipo círculo
            figura = circulo;
```

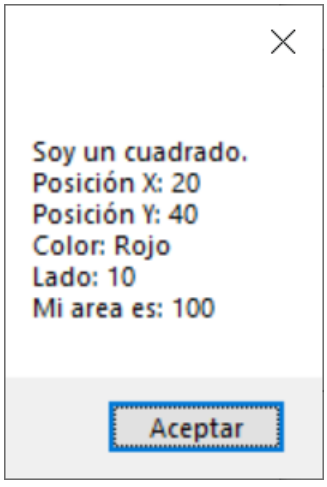
```
            texto = figura.QuienSoy();
            texto = texto + figura.ToString();
            texto = texto + "\nMi area es: " + figura.Area();
            MessageBox.Show(texto);

            // En este momento pasa a referenciar un objeto de tipo cuadrado
            figura = cuadrado;
            texto = figura.QuienSoy();
            texto = texto + figura.ToString();
            texto = texto + "\nMi area es: " + figura.Area();
            MessageBox.Show(texto);
        }
```

El primer mensaje será:



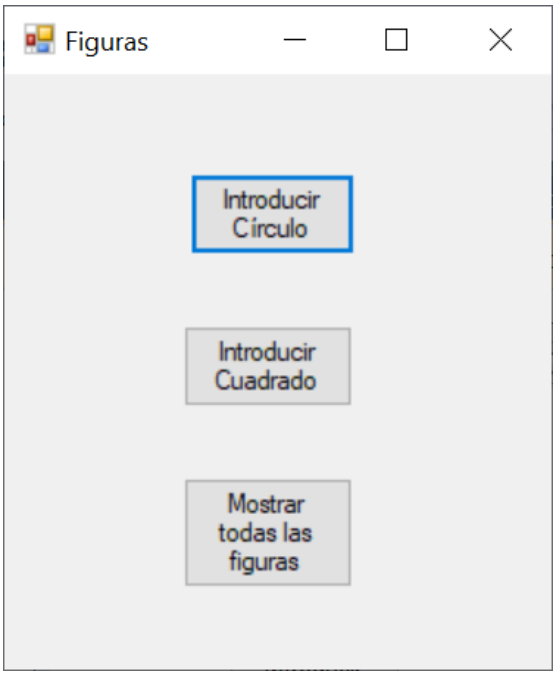
Y el segundo:



Como vemos a pesar de que estamos utilizando una variable de tipo Figura se ejecutan los métodos de Circulo y Cuadrado dependiendo de a qué objeto apunta la figura.

Más Polimorfismo. Ejemplo Final.

Vamos a hacer un ejemplo final en el que entendamos bien una de las ventajas del polimorfismo. Utilizando las clases que hemos definido hasta ahora, vamos a hacer un formulario que nos permita introducir cuadrados y círculos, los introduzca en una lista y luego podamos mostrar las distintas figuras que tenemos.



Tengamos en cuenta que en nuestro proyecto hemos definido, tal y como hemos visto antes, las clases Figura, Circulo y Cuadrado.

Tendríamos definida una lista de figuras y también un método para obtener posiciones y color:

```
// Declaramos una lista donde meteremos TODAS las figuras
List<Figura> listaFiguras = new List<Figura>();

// Subprograma que devuelve posiciones x e y y el color
void obtenerPosicionesYColor(out int posX, out int posY, out string color)
{
    posX = int.Parse(InputBox("Introduzca la posición X"));
    posY = int.Parse(InputBox("Introduzca la posición Y"));
    color = InputBox("Introduzca el color");
}
```

Este sería el código al introducir un círculo:

```
private void bCirculo_Click(object sender, EventArgs e)
{
    int posX, posY, radio;
    string color;

    // Obtenemos los datos del círculo
    obtenerPosicionesYColor(out posX, out posY, out color);
    radio = int.Parse(InputBox("Introduzca el radio del círculo"));

    // Creamos el objeto círculo
    Circulo circulo = new Circulo(posX, posY, color, radio);

    // Lo añadimos a la lista de figuras
    listaFiguras.Add(circulo);
}
```

Este sería el código al introducir un cuadrado:

```
private void bCuadrado_Click(object sender, EventArgs e)
{
    int posX, posY, lado;
    string color;

    // Obtenemos los datos del cuadrado
    obtenerPosicionesYColor(out posX, out posY, out color);
    lado = int.Parse(InputBox("Introduzca el lado del cuadrado"));

    // Creamos el objeto cuadrado
    Cuadrado cuadrado = new Cuadrado(posX, posY, color, lado);

    // Lo añadimos a la lista de figuras
    listaFiguras.Add(cuadrado);
}
```

Y, por último, este es el código que recorre toda la lista:

```
private void bMostrar_Click(object sender, EventArgs e)
{
    int cont = 1;
    // Recorremos toda la lista
    // Utilizamos Figura
    foreach(Figura figura in listaFiguras)
    {
        string texto;

        texto = "Figura num " + cont + "\n";
        texto = texto + figura.QuienSoy() + "\n";
```

```
                texto = texto + figura.ToString() + "\n";
                texto = texto + "Mi área es " + figura.Area();
                        MessageBox.Show(texto);
                cont++;
            }
        }
```

Aquí vemos un ejemplo de utilización del polimorfismo.

El objeto que utilizamos para recorrer los elementos de la lista es de tipo **Figura**, pero cuando instancia o "apunta" a un objeto de un tipo heredado, **los métodos que ejecuta son los de ese tipo heredado**.

Es decir, cuando la figura instancia a un objeto de tipo **Circulo** calcula el área, utilizando el método **Area**() de Circulo que hace 2 * PI * radio.

Igualmente, cuando figura instancia a un objeto de tipo **Cuadrado** utiliza el método **Area**() de Cuadrado que hace lado * lado.

¿Cómo obtener de qué tipo es un objeto?

Una de las cosas que nos puede interesar cuando recorremos una lista de objetos de distinto tipo como ocurre en nuestro caso, es saber de qué tipo es el objeto al que estamos referenciando.

Por ejemplo, nos puede interesar mostrar únicamente los círculos.

O, por ejemplo, acceder a métodos o miembros de los cuadrados como obtener la suma de los lados de todos los cuadrados que tenemos.

Para hacer eso vamos a utilizar el método **GetType** que heredamos de la clase System.Object, que como sabemos es la clase "padre" de todas las clases que declaremos en C#.

Vamos a hacer una función en nuestro formulario que devuelva un texto con todos los datos de los **círculos**:

```
string mostrarDatosCirculos(List<Figura> listaFiguras)
{
    string texto = "Datos de los círculos:\n\n";
    int cont = 1;

    foreach (Figura figura in listaFiguras)
    {
        // Con este if comprobamos si la figura
        // es un círculo

        if (figura.GetType() == typeof(Circulo))
        {
            texto = texto + "Círculo num " + cont + "\n";
            texto = texto + figura.ToString() + "\n";
            texto = texto + "Mi área es " + figura.Area() + "\n\n";
```

```
                cont++;
            }
        }

        return texto;
    }


    private void bMostrarCirculos_Click(object sender, EventArgs e)
    {
        MessageBox.Show(mostrarDatosCirculos(listaFiguras));
    }
```

¿Como podríamos obtener la suma de los perímetros de los cuadrados?…

Para ello podríamos hacer un método **en la clase Cuadrado** que nos devolviera el valor del perímetro de un cuadrado:

```
    public double Perimetro()
    {
        return 4 * mLado;
    }
```

Después en el formulario podríamos hacer una función que recorriera todas las figuras seleccionando los cuadrados y obteniendo la suma de todos los perímetros:

```
    double sumaPerimetrosCuadrados(List<Figura> listaFiguras)
    {
        double suma = 0;

        foreach (Figura figura in listaFiguras)
        {
            if (figura.GetType() == typeof(Cuadrado))
            {

                // Necesario el casting
                suma = suma + ((Cuadrado)figura).Perimetro();
            }
        }

        return suma;
    }
```

Notar que necesitamos hacerle el casting a figura,

```
                suma = suma + ((Cuadrado)figura).Perimetro();
```

para poder acceder al método Perímetro que es un método perteneciente a la clase Cuadrado (y no a las demás).