Encapsulación en C#

Índice

Encapsulación en C#.....1
1 Introducción.....2
2 Propiedades.....4
2.1 Acceso a los atributos.....4
2.2 Definición abreviada de propiedades y atributos.....6
2.3 Ahondando en las propiedades de solo lectura.....8
2.4 El papel de las propiedades en el polimorfismo.....9
3 Publicando vectores, matrices o colecciones.....11

1 Introducción

En una clase típica, se plantean atributos, que almacenarán el estado del objeto, y unos métodos para acceder a ellos, tan comunes, que se denominan *getters* y *setters* (permiten obtener los valores de los atributos, y modificarlos, respectivamente). Por ejemplo:

```
class Persona {
      public Persona(string n, string e)
      {
            this.nombre = n;
            this.email = e;
      }

      string email;                 // privado por defecto
      string nombre;
};
```

Aquellos miembros de la clase que no tienen un modificador de visibilidad (*private*, *protected*, *public*), son considerados automáticamente privados. A partir de este momento, sería necesario crear los citados métodos *getters* y *setters*.

```
class Persona {
      public Persona(string n, string e)
      {
            this.nombre = n;
            this.email = e;
      }

      public string GetEmail()
      {
            return this.email;
      }

      public string GetNombre()
      {
            return this.nombre;
      }

      public void SetNombre(string n)
      {
            this.nombre = n;
      }

      public void SetEmail(string e)
      {
            this.email = e;
      }

      string email;                 // privado por defecto
      string nombre;
};
```

Intuitivamente, aquellos miembros marcados como públicos (*public*), son accesibles desde el exterior, mientras que aquellos marcados como privados (*private* o sin modificador de visibilidad), solo son accesibles desde dentro de la propia clase. El modo protegido (*protected*), es

específico de casos en los que se presenta herencia entre clases. Así, si la clase **Empleado** hereda de **Persona**, un método o atributo *protected* será accesible desde la propia clase **Persona** y en clases que hereden de ella, como **Empleado**, mientras se considerará privado en el resto de situaciones.

```
class Persona {
      public Persona(string nif, string n, string e)
      {
            this.nif = nif;
            this.nombre = n;
            this.email = e;
      }

      public string GetNif()
      {
            return this.nif;
      }

      public string GetEmail()
      {
            return this.email;
      }

      public string GetNombre()
      {
            return this.nombre;
      }

      public void SetNombre(string n)
      {
            this.nombre = n;
      }

      public void SetEmail(string e)
      {
            this.email = e;
      }

      string email;                 // privado por defecto
      string nombre;
      protected string nif;
};
```

No se debe abusar del acceso protegido (protected), prefiriendo el acceso privado (probablemente con un *getter* pero sin un *setter*), siempre que sea posible.

Dado que la creación de métodos *getter* y *setter* resulta ser un trabajo muy repetitivo, resulta pesado y bastante propenso a cometer errores. Como se verá a continuación, las propiedades surgen como respuesta a este problema.

2 Propiedades

Las propiedades surgen como solución a este problema. Se trata de aparentes atributos, que en realidad, cuando se lee de ellos, están devolviendo el atributo real, y cuando se modifican, están accediendo al atributo real.

```
class Persona {
      public Persona(string n, string e)
      {
            this.nombre = n;
            this.email = e;
      }

      public string Email {
            get { return this.email; }
            set { this.email = value; }
      }

      public string Nombre {
            get { return this.nombre; }
            set { this.nombre = value; }
      }

      string email;
      string nombre;
}
```

La convención de escritura de código consiste en mantener los atributos en minúsculas, y las propiedades en mayúsculas.

La pregunta es, en este momento, que si realmente es una equiparación directa entre un método escondido, y el atributo real, no parece haber motivación para tomarse la molestia. Efectivamente,

```
class Ppal {
      public static void Main()
      {
            var p = new Persona(
                  "Baltasar García", "jbgarcia@uvigo.es" );

            p.Nombre = "Baltasar García Perez-Schofield";
            System.Console.Write(
                        p.Nombre, p.Apellido1, p.Apellido2
            );
      }
}
```

2.1 Acceso a los atributos

Aunque hasta ahora se ha planteado el que exista una propiedad por cada atributo con funciones *get* y *set*, en realidad, esto no tiene por qué ser así. De la misma forma que no todos los atributos tienen por qué estar expuestos (aunque también hay que tener en cuenta que las propiedades pueden ser públicas, privadas o protegidas), una propiedad no tiene que proveer siempre de un *get* y un *set*; las propiedades que no tengan este último serán de sólo lectura.

Por ejemplo, en la clase **Persona**:

```
class Persona {
      public Persona(string n, string e)
      {
            this.nombre = n;
            this.email = e;
      }

      public string Email {
            get { return email; }
            set { email = value; }
      }

      public string Nombre {
            get { return nombre; }
            set { nombre = value; }
      }

      public string Apellido1{
            get { return Nombre.Split()[ 1 ]; }
      }

      public string Apellido2{
            get { return Nombre.Split()[ 2 ]; }
      }

      string email;
      string nombre;
}
```

En el código anterior, las propiedades *Apellido1* y *Apellidos2* llaman al método *Split()* de las cadenas para dividir el nombre, usando el carácter espacio, que presumiblemente habrá entre el nombre, el primer apellido, y el segundo (en caso de que no se cumplan estas asunciones, se producirá una excepción en tiempo de ejecución).

En cualquier caso, la gran ventaja de las propiedades es que presentan la oportunidad de transformar el contenido de los atributos sin tener que cambiar estos: suponiendo que las especificaciones para una clase cambien, es posible no tener que cambiar la clase (y, por tanto, todos sus datos), utilizando la oportunidad de las propiedades para transformar los valores. Por ejemplo, en el siguiente código, una antigua clase **Cuenta** que almacenaba el saldo en las antiguas pesetas puede cambiarse a euros fácilmente.

```
class Cuenta {
      int saldo;

      public double Saldo {
            get { return this.saldo / 166.386; }
            set { this.saldo = value * 166.386; }
      }
}
```

2.2 Definición abreviada de propiedades y atributos

La única precondición es acostumbrarse a utilizar las propiedades (bien sean de sólo lectura, o de lectura y escritura) con todos los atributos. Para ello, *C#* proporciona un mecanismo muy cómodo, de tal manera que la definición de una propiedad con métodos *get* y *set* por defecto implica la creación del atributo automáticamente. Así, por ejemplo, la antigua definición de la clase **Cuenta** podría haberse creado en su día como sigue:

```
class Cuenta {
      public double Saldo { get; set; }
}
```

Ya no es necesario crear el atributo *saldo*, pues la propiedad por defecto *Saldo* implica que *C#* creará automáticamente el atributo.

Una pequeña variante es conseguir una propiedad que será de solo lectura, o dicho de otra manera, se podrá leer pero no escribir (excepto dentro de la propia clase).

```
class Persona {
      public Persona(int dni) {
            this.Dni = dni;
      }

      public int Dni {
            get; private set;
      }

      public string Nombre {
            get; set;
      }

      public string Apellidos {
            get; set;
      }
}
```

De la misma forma, la clase **Persona** se podría reescribir de la siguiente forma, más compacta:

```
class Persona {
      public Persona(string n, string e)
      {
            Nombre = n;
            Email = e;
      }

      public string Email {
            get; set;
      }

      public string Nombre {
            get; set;
      }

      public string Apellido1{
            get { return Nombre.Split()[ 1 ]; }
      }

      public string Apellido2{
            get { return Nombre.Split()[ 2 ]; }
      }
}
```

Y si deseamos ir un poco más allá en cuanto a abreviar, es posible utilizar la sintaxis => para indicar la expresión devuelta por un método o un **get** o **set** de una propiedad.

```
class Persona {
      public Persona(string n, string e)
      {
            Nombre = n;
            Email = e;
      }

      public string Email {
            get; set;
      }

      public string Nombre {
            get; set;
      }

      public string Apellido1{
            get => Nombre.Split()[ 1 ];
      }

      public string Apellido2{
            get => Nombre.Split()[ 2 ];
      }
}
```

2.3 Ahondando en las propiedades de solo lectura

Una propiedad de solo lectura, como se indicaba más arriba, puede definirse con su parte de escritura (*set*) como privada, o simplemente puede faltar. En el primer caso, la propiedad solo puede modificarse desde un método privado de la clase. En el último caso en cambio, esto indica que la propiedad solo puede modificarse desde el constructor (siempre y cuando no sea una propiedad "calculada" como en el caso anterior en *Apellido1* y *Apellido2*).

```
class Persona {
      public Persona(string n, string e)
      {
            this.Nombre = n;
            this.Email = e;

            this.Apellido1 = this.Nombre.Split()[ 1 ];
            this.Apellido2 = this.Nombre.Split()[ 2 ];
      }

      public string Email {
            get;
      }

      public string Nombre {
            get;
      }

      public string Apellido1{
            get;
      }

      public string Apellido2{
            get;
      }
}
```

En ese último ejemplo, un objeto de la clase **Persona** es inmutable, por lo que tiene sentido precalcular *Apellido1* y *Apellido2*, de manera que ya estén listas para su uso en cuanto el objeto haya sido creado.

Si aceptamos que el nombre o el apellido pueden cambiar, pero deseamos precalcular *Apellido1* y *Apellido2*, entonces tendremos un escenario para propiedades de solo lectura con *private set*. Eso sí, como es necesario realizar operaciones adicionales después de asignar el *Nombre*, ya no es posible utilizar la definición abreviada para ella.

```
class Persona {
      public Persona(string n, string e)
      {
            this.Nombre = n;
            this.Email = e;
      }

      public string Email {
            get; set;
      }

      public string Nombre {
            get {
                  return this.email;
            }
            set {
                  this.email = value;
                  this.Apellido1 = this.Nombre.Split()[ 1 ];
                  this.Apellido2 = this.Nombre.Split()[ 2 ];
            }
      }

      public string Apellido1{
            get; private set;
      }

      public string Apellido2{
            get; private set;
      }

      string nombre;
}
```

2.4 El papel de las propiedades en el polimorfismo

Las propiedades contienen al menos una función. Debido a que las funciones pueden llevar los modificadores *abstract*, *override*, y *new*.

Un ejemplo típico es de las figuras, en la que distintas figuras geométricas heredan de una clase **Figura**, que define un método *calculaArea()*, o similar, como *abstract*, ya que será definido en realidad en las clases derivadas. Ese método abstracto permite conocer que todas las clases derivadas de **Figura** van a tenerlo.

```
class Figura {
      public abstract double Area {
            get;
      }
}
```

Todas las clases que deriven de Figura, deberán por tanto definir esta propiedad *Area*, y para poder hacerlo es necesario redefinirla utilizando el modificador *override*. Nótese que es posible crear una nueva propiedad *Area* que rompa con el polimorfismo definido mediante la herencia desde la clase base. En tal caso, será necesario utilizar el modificador *new* para la propiedad.

A continuación muestran las clases Rectángulo y Círculo.

```
class Rectangulo {
	public Rectangulo(double l1, double l2)
	{
		this.Lado1 = l1;
		this.Lado2 = l2;
	}

	public override double Area {
		get { return this.Lado1 * this.Lado2; }
	}

	public double Lado1 {
		get; set;
	}

	public double Lado2 {
		get; set;
	}
}

class Circulo {
	public Circulo(double r)
	{
		this.Radio = r;
	}

	public override double Area {
		get { return this.Radio * this.Radio * Math.PI; }
	}
}
```

3 Publicando vectores, matrices o colecciones

Es necesario ser muy cuidadoso a la hora de mantener un acceso público a un vector (*array*), o colección (**List**<>, **Set**<>, etc.). Mientras que dar acceso mediante una propiedad *get* efectivamente impide la modificación de un tipo de dato básico, como un entero, real o cadena de caracteres, esto no sucede así cuando se trata de un vector o en general de cualquier colección de valores.

```
class Punto {
      public Punto(double x, double y)
      {
            this.X = x;
            this.Y = y;
      }

      public double X { get; }
      public double Y { get; }

      public double DistanciaCon(Punto p)
      {
            return Math.Sqrt( Math.Pow( this.X – p.X, 2 ),
                              Math.Pow( this.Y – p.Y, 2 ) );
      }

      public override string ToString() => $”{this.X}, {this.Y}”;
}
```

La siguiente clase **Figura** utiliza una colección List<Punto> para almacenar los puntos que conforman el perímetro de la misma. La colección List<> almacena valores primitivos (enteros, reales, cadenas de caracteres), u objetos, sin que el programador deba preocuparse de expandir la colección cuando se queda sin memoria. El método **List**<>.Add(x) permite añadir nuevos elementos, mientras que la propiedad *Count* devuelve el número de elementos almacenados.

```
class Figura {
	public Figura()
	{
		this.puntos = new List<Punto>();
	}

	public void Add(Punto p)
	{
		this.puntos.Add( p );
	}

	public override double Perimetro {
		get {
			double toret = 0;
			Punto anterior = this.puntos[ 0 ];

			for(int i = 1; i < this.puntos.Count; ++i) {
				var actual = this.puntos[ i ];

				toret += anterior.DistanciaCon( actual );
				anterior = actual;
			}

			return toret;
		}
	}

	public List<Punto> Puntos {
		get { return this.puntos; }
	}

	List<Punto> puntos;
}
```

Claramente, el autor de esta clase no desea dar acceso a los puntos de la figura (de otra forma, seguramente habría un *set* en la propiedad *Puntos*). Sin embargo, al devolver directamente el atributo *puntos*, es posible modificar la figura desde fuera de la clase.

```
var f = new Figura();

f.Add( new Punto( 5, 6 ) );
f.Add( new Punto( 7, 8 ) );

f.Puntos.RemoveAt( 0 ); // Elimina el primer punto de la figura !!
```

Así, debe evitarse siempre devolver directamente vectores y colecciones. En su lugar, si realmente es necesario dar acceso a estos elementos, se puede devolver una copia, por poner un ejemplo.

Se puede copiar un vector mediante el método **Array**.*CopyTo(arrayDest, indiceComienzo)*, mientras que cualquier colección tiene el conveniente ToArray(), o puede crearse otra colección directamente pasándola en el constructor.

```
// Copiando un array
var a1 = new int[]{ 1, 2, 3, 4, 5 };
var a2 = new int[ a1.Length ];

a1.CopyTo( a2, 0 );

// Copiando una colección
var l1 = new List<int>();

l1.AddRange( new []{ 1, 2, 3, 4, 5 } );

int[] a3 = l1.ToArray();            // Copiando l1 en array a3

var l2 = new List<int>( l1 );       // Copiando list l1 en l2
```

Así, la clase Figura podría quedar como sigue:

```
class Figura {
      public Figura()
      {
            this.puntos = new List<Punto>();
      }

      public void Add(Punto p)
      {
            this.puntos.Add( p );
      }

      public override double Perimetro {
            get {
                  double toret = 0;
                  Punto anterior = this.puntos[ 0 ];

                  for(int i = 1; i < this.puntos.Count; ++i) {
                        var actual = this.puntos[ i ];

                        toret += anterior.Distancia( actual );
                        anterior = actual;
                  }

                  return toret;
            }
      }

      public Punto[] Puntos {
            get { return this.puntos.ToArray(); }
      }

      List<Punto> puntos;
}
```