Tema 9.2

Descargar estos apuntes

Índice

Profundizando en la Programación Orientada a Objetos

Diferentes Polimorfismos Diferentes Formas

Definiciones de polimorfismo

1. **Polimorfismo datos o inclusión**
   - **Ya lo hemos visto** cuando estudiamos el concepto de herencia y downcasting. Se basa en el Upcasting o **Principio de Sustitución de Liskov**
   - Es de datos porque, tenemos un objeto o dato con diferentes formas dependiendo del tipo con que lo referenciemos.
   - Además de este, tendremos otras formas de polimorfismo...
2. **Polimorfismo paramétrico o tipos genéricos**
   - Lo vamos a tratar más adelante en este tema 9.
3. **Polimorfismo funcional o sobrecarga**
   - Al igual que el de datos, **ya lo hemos visto** y usado con anterioridad. Pero ahora es cuando lo definiremos formalmente como una característica de los lenguajes OO.
   - Es la **capacidad de definir operaciones o métodos con el mismo identificador** o nombre. Siempre y cuando, la signatura cambie.
   - Recuerda que en C# dos métodos tienen diferente signatura si:
     - Tienen diferente tipo de retorno.
     - Tienen diferente número de parámetros.
     - Teniendo el mismo número de parámetros y algún tipo es diferente.
     - Teniendo el mismo número de parámetros y el mismo tipo alguno tiene el modificador ref o out.

```
Class SobrecargaValida
{
    public void MetodoA(int x) { ; }
    public void MetodoA(ref int x) { ; }
}
```

```
Class SobrecargaInvalida
{
    public void MetodoA(out int x) { ; }
    public void MetodoA(ref int x) { ; }
}
```

- **¿Para qué se usa el polimorfismo funcional?**
  - Por ejemplo, lo venimos cuando tuvimos que **definir varios constructores** en una clase.
  - Para **evitar el uso de parámetros opcionales o por defecto** en los métodos.

Parámetros opcionales o por defecto

Una llamada a un método debe proporcionar los argumentos reales para todos los parámetros, sin embargo **se pueden omitir aquellos argumentos** de parámetros opcionales.

✋ Los parámetros opcionales **se definen al final de la lista de parámetros** después de los parámetros necesarios. Si el autor de la llamada proporciona un argumento para algún parámetro de una sucesión de parámetros opcionales, **debe proporcionar argumentos para todos los parámetros opcionales anteriores** o en su lugar indicar el identificador del parámetro formal.

Veamos esto último a través de un **ejemplo**, de sintaxis:

```
    static class Ejemplo {
        static void Metodo(
                string cadenaRequerida, int enteroRequerido,    // No puedo definir ningún op
                string cadenaOpcional = "", int enteroOpcional = 10)
                { ... }
        static void Main() {
            Metodo("Cadena obligatoria", 3, "Cadena Opcional", 33); // Correcto
            Metodo("Cadena obligatoria", 3, "Cadena Opcional");     // Correcto enteroOpciona
            Metodo("Cadena obligatoria", 3);                        // Correcto cadenaOpciona
            // Si sabemos el nombre del identificador del parámetro en el método...
            Metodo("Cadena obligatoria", 3, enteroOpcional: 10);    // Correcto cadenaOpciona

            Metodo("Cadena obligatoria", 3, 10);                    // Incorrecto
            Metodo("Cadena obligatoria", 3, , 10);                  // Incorrecto
        }
    }
```

Se pueden definir en multitud de lenguajes como C#, Python, PHP, Javascript, Kotlin, etc. Sin embargo **Java no los permite** porque tienen inconvenientes:

- 💀 Mal usados, **pueden dar lugar a baja cohesión** (métodos '*navaja suiza*' o que hacen muchas cosas según los parámetros que le lleguen).
- Ralentizan la ejecución.
- Lleva a confusión a los usuarios de una clase.

> ✋ **Importante:** Por las razones anteriores. **No deberíamos usarlos en métodos públicos**.
> (👁 *fíjate que Microsoft apenas los usa en sus BCL y sí la sobrecarga*)

Caso de estudio:

Vamos a tratar un ejemplo de como evitar parámetros opcionales en los métodos públicos o **en lenguajes que no nos los permitan como Java** a través de C#.

Si recordamos de temas anteriores, definimos una estructura `Punto2D` que ahora va a tener ahora el método `Desplaza` con el valor del ángulo a 0 de forma opcional.

```
struct Punto2D
{
    public readonly double X;
    public readonly double Y;

    public Punto2D(in double x, in double y) { Y = y; X = x; }

    public Punto2D Desplaza(in double distancia, double anguloGrados = 0D)
    {
        double anguloRadianes = anguloGrados * Math.PI / 180d;
        double fila = Y + distancia * Math.Sin(anguloRadianes);
        double columna = X + distancia * Math.Cos(anguloRadianes);
        return new Punto2D(fila, columna);
    }

    public override string ToString() { return $"({X:G2} - {Y:G2})"; }
}
```

Ahora en un programa podríamos instancias un objeto valor Punto2D

`Punto2D p = new Punto2D(2D, 4D);`

y continuación hacer ...

`p.Desplaza(4D);`

Como el parámetro formal `anguloGrados` es opcional, podremos llamar al método `Desplaza` sin especificarlo y en ese caso caso tomará su valor por defecto `0D` grados, desplazando el punto 4 unidades a la derecha.

Caso de estudio (continuación...)

¿Cómo deberíamos refactorizar el código anterior **usando polimorfismo funcional** o sobrecarga?

La forma más común sería la siguiente...

```
struct Punto2D
{
    // Definimos como privado el método a sobrecargar para no repetir el código
    // además para que no haya conflicto de nombres le ponemos un _
    private Punto2D _Desplaza(in double distancia, double anguloGrados)
    {
        double anguloRadianes = anguloGrados * Math.PI / 180d;
        double fila = Y + distancia * Math.Sin(anguloRadianes);
        double columna = X + distancia * Math.Cos(anguloRadianes);
        return new Punto2D(fila, columna);
    }

    // Definimos las sobrecargas públicas, con los parámetros posibles.
    public Punto2D Desplaza(ushort numPosiciones)
    {
        // Aquí decidiremos el valor por defecto.
        return _Desplaza(numPosiciones, 0d);
    }

    public Punto2D Desplaza(ushort numPosiciones, double angulo)
    {
        return _Desplaza(numPosiciones, angulo);
    }
}
```

EN CONSTRUCCION PROPIEDADES

DE LA PRESENTACION

Encapsulación
Objetivos
• Evitar que un cliente de mis clases puedan dejar objetos instanciados de las mismas en un estado inadecuado.
• Ocultar detalles de la implementación de una clase.
• Realizar cambios o actualizaciones en la clases sin preocuparnos cómo están siendo usadas.
• Ya hemos usado la encapsulación, marcando los atributos como privados y a través del uso de accesores y mutadores.

Encapsulación
Recordemos los modificadores de acceso que hay para clases, tipos, atributos y métodos.
• private: Accesible solo desde la clase. Es lo que debemos poner por defecto.
• public: Accesible por todos.
• protected: Accesible solo desde la clase o las subclases.
• internal: Accesible solo desde clases del ensamblado o paquete actual.
• protected internal: Accesible solo desde clases del paquete actual y además sean subclases de la clase donde se ha definido.
Encapsulación
Propiedades en C# - I
• Hasta ahora hemos utilizado la sintaxis de otros lenguajes como Java, C++ , php, etc… para definir los accesores y mutadores.

```
get() {
return
}
void set( ) {
this. =
}
```

```
• Las propiedades son un "azucar sintáctico" incluido por el lenguaje C# para usarlos y de es
```

```
{
set {
= value;
```

```
}
get {
return ;
}
}
```

Encapsulación

Propiedades en C# - II

• Por ejemplo, para nuestro atributo fila en la clase PuntoConsola hacíamos….

```
public void SetFila(ushort fila) {
if (fila > 24)
throw new ArgumentException("fila > columna 24");
this.fila = fila;
}
private ushort getFila() {
return fila;
}
```

• Si los definimos como una propiedad en C# haremos…

```
public ushort Fila {
set {
if (value > 24)
throw new ArgumentException("fila > columna 24");
fila = value;
}
private get {
return fila;
}
}
```

Encapsulación

Propiedades en C# - III

• Podremos definir solo uno de los dos, set o get y estarán afectadas por los modificadores de accesibilidad como el resto de métodos.

¿ Cómo usaremos la propiedad cuando instanciemos un objeto de la clase PuntoConsola ?

• Para nosotros sintácticamente, es como si estuvieramos accediendo directamente al atributo.

```
PuntoConsola punto = new PuntoConsola(2, 4);

Console.WriteLine(punto.Fila); // Solo se podrá acceder a la
// propiedad Fila dentro de la clase.
// Por estar marcado el set como private.
```

punto.Fila = 12; // Cambiaremos el valor de la fila a 12.
// Internamente se llama al set

punto.Fila = 25; // Generará un ArgumentException

Encapsulación
Propiedades autoimplementadas a partir de C# 3.0 - IV
• A partir de C# 3.0 , aparecen las propiedades autoimplementadas que hacen que la declaración de propiedad sea más concisa cuando no se requiere ninguna lógica adicional en los descriptores de acceso de la propiedad.
• https://msdn.microsoft.com/es-es/library/bb384054.aspx

DE LOS APUNTES DE XUSA

Una propiedad es una mezcla entre el concepto de campo y el concepto de método. Externamente es accedida como si de un campo normal se tratase, pero internamente es posible asociar código a ejecutar en cada asignación o lectura de su valor. Es la forma de implementar los métodos de acceso (accesores) y actualización (mutadores), que permiten trabajar de forma segura con los atributos privados en c#.

```
         Su sintaxis es la siguiente:
```

{
set
{
Atributo = value;
}
get
{
return Atributo;
}
}
set Es un método de actualización y el valor del atributo nos llega en un parámetro especial denominado value. get Es un método de acceso. En una propiedad al menos uno de los dos debe estar definido. Es decir, podremos definir solo uno de los dos, set o get y estarán afectadas por los modificadores de accesibilidad como el resto de métodos.

NOMENCLATURA: Si el nombre del atributo coincide con la propiedad, el atributo irá en camel Casing y puede ir precedido del carácter '_' y la propiedad toma el nombre del atributo en Pascal Casing.

Ejemplo:
Para acceder o cambiar el atributo Fila de nuestra clase PuntoConsola con una propiedad, en primer lugar cambiaremos el nombre del atributo Fila por fila o por _fila. Después añadiremos la siguiente propiedad:

```
public ushort Fila
{
set { if(value <= 24) fila = value;}
get { return fila; }
}
```

Si en el Main() ponemos:
Console.WriteLine(punto.Fila); // Veremos el valor de la fila. Internamente se llama al get
punto.Fila = 12; // Cambiaremos el valor de la fila a 12. Internamente se llama al set

En C# 3.0 y versiones posteriores, aparecen las propiedades autoimplementadas que hacen que la declaración de propiedad sea más concisa cuando no se requiere ninguna lógica adicional en los descriptores de acceso de la propiedad. Al declarar una propiedad tal y como se muestra en el ejemplo siguiente, el compilador crea un campo de respaldo privado y anónimo al que solamente puede obtenerse acceso a través de los descriptores de acceso get y set de la propiedad.

```
class Trabajo
{
// Propiedades autoimplementables
public double SueldoTotal { get; set; }
public string Nombre { get; set; }
public int Identificador { get; set; }

    // Constructor
    public Trabajo(double sueldo, string nombre, int id)
    {
       this.SueldoTotal= sueldo;
       Nombre = nombre;
       this.Identificador = id;
    }

}

class Program
{
```

```
    static void Main()
    {
        Trabajo trabajador = new Trabajo(4987.63, "Alicante",
                                         90108);
        trabajador.SueldoTotal += 499.99;
    }
}
```